Time-triggered Runtime Verification of Real-time Embedded Systems

by

Samaneh Navabpour

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2014

This thesis consists of material all of which I authored or co-authored: See Statement of Contributions included in this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In safety-critical real-time embedded systems, correctness is of primary concern, as even small transient errors may lead to catastrophic consequences. Due to the limitations of well-established methods such as verification and testing, recently runtime verification has emerged as a complementary approach, where a monitor inspects the system to evaluate the specifications at run time. The goal of runtime verification is to monitor the behavior of a system to check its conformance to a set of desirable logical properties. The literature of runtime verification mostly focuses on *event-triggered* solutions, where a monitor is invoked when a significant event occurs (e.g., change in the value of some variable used by the properties). At invocation, the monitor evaluates the set of properties of the system that are affected by the occurrence of the event. This type of monitor invocation has two main runtime characteristics: (1) jittery runtime overhead, and (2) unpredictable monitor invocations. These characteristics result in transient overload situations and over-provisioning of resources in real-time embedded systems and hence, may result in catastrophic outcomes in safety-critical systems.

To circumvent the aforementioned defects in runtime verification, this dissertation introduces a novel *time-triggered* monitoring approach, where the monitor takes samples from the system with a constant frequency, in order to analyze the system's health. We describe the formal semantics of time-triggered monitoring and discuss how to optimize the sampling period using minimum auxiliary memory and path prediction techniques. Experiments on real-time embedded systems show that our approach introduces bounded overhead, predictable monitoring, less over-provisioning, and effectively reduces the involvement of the monitor at run time by using negligible auxiliary memory. We further advance our time-triggered monitor to component-based multi-core embedded systems by establishing an optimization technique that provides the invocation frequency of the monitors and the mapping of components to cores to minimize monitoring overhead. Lastly, we present RiTHM, a fully automated and open source tool which provides time-triggered runtime verification specifically for real-time embedded systems developed in C.

Acknowledgements

This is an opportunity for me to express my gratitude towards everyone who has supported me throughout the course of my PhD. First and foremost, I would like to thank my supervisor, Professor Sebastian Fischmeister, for his encouragement, support and guidance. His advice and outlook guided my research towards the right direction and his compassion for my well-being played an important role in my life.

I would like to thank Professor Borzoo Bonakdarpour for his passionate guidance and support at every turn in my research. His enthusiasm helped me overcome many obstacles throughout my studies. I would also like to thank my dissertation committee: Professor Lin Tan and Professor Ondřej Lhoták for taking the time and effort to participate in my committee, read my dissertation, and provide me with helpful and valuable feedback throughout the course of my research; Professor Derek Rayside for his constant support all through my studies and his invaluable guidance and feedback on my research; and Professor Scott Smolka, my external examiner, for putting aside the time to read my dissertation despite his busy schedule and providing me with insightful comments.

I would like to specially thank Adam Mallory and Chris Hobbs for providing me with the chance to experience a life changing opportunity at QNX. I would specifically like to thank Chris Hobbs for his invaluable mentorship and insightful training and guidance, and becoming the role model that I deeply needed for years. I would like to thank Patrick Lee for being the best colleague anyone could ask for, his kindness, support, and advice helped me through challenging times. I would also like to specially thank Nicola Vulpe, Gianluca Ragazzini, and Steve Reid for their invaluable friendship, rejuvenating company, and helpful advice. Our lunches will be greatly missed.

A huge thanks to all my friends in the Real-time Embedded Systems group that stood beside me both in laughter and tears, and creating unforgettable and irreplaceable memories. I am extremely fortunate to have met, befriended and worked with these amazing people: Johnson Thomas, Wallace Wu, Hany Kashif, Ramy Medhat, Augusto Born de Oliveira, Pansy Arafa, Akramul Azim, Sina Gholamian, and Shay Berkovich.

I am deeply thankful to my dear parents, Hamidreza and Mahshid, for their support throughout every step of my life. I specially want to thank my mother, for her unlimited sacrifices she has made to stand beside me with her head held high, even when all odds were against me. I can not imagine where I would be without her. I would also like to thank my beautiful sister, Sanieh, for being my soul-mate and representing everything that is good in this world. Big thanks to my strong-headed brother, Alireza, for reminding me everyday that nothing in this world is worth giving up on your dreams. In addition, I

would like to thank Danielle Tyldsley for her invaluable friendship and providing me with a home away from home.

Last, but not least, my profound appreciation and love goes to my beloved husband, Ali. There is no doubt in my mind that this work would have never been accomplished without his constant support and sacrifice. I genuinely appreciate his encouragement and unlimited patience and all the happiness he has brought into my life.

Dedication

My beloved husband, Ali,

for his constant love, support and patience.

My dear mother, Mahshid,

for her endless sacrifices and making me the person I am today.

My lovely sister, Sanieh,

for being my rock and ray of sunshine in every moment of everyday.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Program correctness is one of the greatest issues facing software developers today. A recent NIST report estimates that $59.6 billion dollars, or 0.6% of the GDP, are lost every year because of software errors [88] [69]. The common approach taken by developers to ensure program correctness is either *program verification* or *testing*.

The field of program verification such as model checking and theorem proving, tackles the problem of determining whether a program satisfies a set of properties (i.e., specifications). Program verification aims at proving that *all* execution paths of the program satisfy the properties. Although program verification is a well studied approach which is automated and exhaustively studies the system under inspection, unfortunately, program verification requires developing a rigorous abstract model of the system and suffers from the state-explosion problem [20].

Testing has been the main approach to ensure correctness in software systems. This approach has a long history and has been well studied and hence, the software community has seen significant improvement and development in testing. On the other hand, in the recent decade, we have seen a steep rise in the complexity of individual software components and as a result, the number of lines of code, and the number of execution paths of a software system have drastically increased over time [40] [41]. To this end, testing all possible execution paths of such systems endures high costs. As a result, it is impractical to test all the possible execution paths of a system before deployment. In addition, it is impossible to account for all the possible abnormal behavior from the underlying software and hardware of the system under inspection. Such abnormal behavior can be caused by heat, cosmic rays, bugs in the operating system, human error, etc. Hence, statically based testing techniques cannot examine these programs in their actual execution environments.

The limitations of program verification and testing argue for techniques that can check the correctness of the program at run time. This results in the complementary technique of *Runtime Verification* [75] [49] [10] [8]. Runtime verification is a less ambitious, but more feasible approach. It only aims to prove that the execution path of a program at run time satisfies a set of properties.

Runtime verification incorporates a *monitor*. In the literature of runtime verification, constructing a monitor involves synthesizing an automaton that realizes the properties that the program must satisfy [58]. Then, by composing the monitor with the program, the monitor observes the occurrence of each transition and decides whether the properties have been met, violated, or impossible to tell. Thus, every *significant event* (i.e, events that may change the evaluation of properties at run time) executed in the program, invokes the monitor. We refer to this type of monitoring as *event-triggered*. Deploying an event-triggered monitor involves instrumenting the program, so that upon occurrence of significant events, the instrumentation instruction invokes the monitor to re-evaluate the property. The main drawback of event-triggered monitoring is twofold: the monitor (1) has jittery monitoring overhead (i.e., the variance of the monitoring overhead is large); meaning that the overhead imposed by the monitor from one invocation to another severely differ, and (2) the invocations of the monitor are unevenly distributed throughout the system run. To this end, the event-triggered monitor imposes unpredictable overhead and may introduce bursts of interruptions to the program at run time. These side affects can lead to undesirable transient overload situations in time-sensitive systems such as safety-critical real-time embedded systems.

With this motivation, in this dissertation, we aim at establishing a runtime verification framework suitable for safety-critical real-time embedded systems. This runtime verification framework must have the following runtime characteristics:

1. *predictable monitoring*; in this desertion, we consider a monitoring behavior to be predictable when the monitor invocations are evenly distributed throughout the system run,

2. *bounded monitoring overhead*; in this dissertation, we consider a monitor to have bounded overhead when the overhead imposed by the monitor at each invocation is equal to a constant value (with a jitter equal to a predefined epsilon is acceptable),

3. *efficient monitoring overhead*; in this dissertation, we consider a monitor to have efficient overhead when the monitor does not saturate the system resources.

4. *reduce over-provisioning*; in this dissertation, we aim at reducing the amount of resources reserved for the monitor.

To create a monitor with the aforementioned characteristics, we establish an alternative and novel approach where the monitor is *time-triggered* [21] [19]. The idea is that the monitor wakes up with a constant frequency and takes samples (i.e., extracts state information regarding the affects of the executed significant events) from the program in order to evaluate the properties. This way, the monitor invocations are predictable. In addition, since the time-triggered monitor extracts the same set of state information at each invocation, it imposes an equivalent amount of overhead at each invocation, resulting in bounded monitoring overhead.

The main challenge in creating a time-triggered monitor is accurate reconstruction of the program's state between two samples; i.e., tracking the execution of all the significant events. When the monitor misses a significant event, it may fail to detect violations of some properties. Hence, the problem boils down to finding the longest possible sampling period that allows state reconstruction. We propose a static approach to calculate this sampling period. Our *static* approach [21] calculates the sampling period through building the program's control-flow graph, locating the significant events in the system, and extracting the best case execution time of the system's instructions. We use this sampling period to construct the time-triggered monitor using timed automata. We prove that our calculations are sound and complete and hence, result in sound verification of the system at run time.

The sampling period extracted from the static approach is conservative, since we are oblivious of the execution path of the program at run time. Hence, this sampling period precipitates highly frequent invocations of the time-triggered monitor even in portions of the system that do not require monitoring. Experimental results from our time-triggered monitor show that on average, it imposes 170% monitoring overhead [21] [19]. On the other hand, studies show that in embedded systems, monitoring overhead of over 10% is regarded as unacceptable [33], specifically in real-time embedded systems. This results in the inapplicability of the time-triggered monitor.

To address the above challenge, there are well-established optimization approaches designed for event-based monitors. Some approaches carry out static analysis to eliminate unnecessary instrumentation instructions with respect to the verification of the properties [15] [35]. Another set of approaches use the execution path and the state (i.e., verified/violated) of properties to activate and deactivate instrumentation instructions [29] [76] [34]. In addition, there are approaches which sample a subset of the instrumentation instructions. In other words, throughout the program run, these approaches select a subset of the significant events to monitor [5] [38] [44].

All the above optimization approaches rely on the instrumentation of significant events in the program, the technique used by event-triggered monitoring to observe significant

events. In other words, such instrumentation and optimization techniques are unsuitable for real-time embedded systems, since time-triggered monitor does not use instrumentation. To this end, this dissertation presents optimization techniques to reduce the monitoring overhead of the time-triggered monitor. We present the following two main optimization techniques.

1. *History approach* [21] [19] [85]: This approach increases the sampling period (i.e., reduces the number of samples) by incorporating auxiliary memory, where it stores a history of state changes in the program. In other words, at a subset of sampling points, this approach stores state information related to the executed significant events in an auxiliary memory, instead of the monitor taking a sample. In this case, at each remaining sampling point, the monitor not only samples the current state information, it also extracts the information stored in the auxiliary memory. Hence, the monitor keeps track of every executed significant event while reducing the number of samples. Obviously, this approach faces a tradeoff between minimizing the size of auxiliary memory versus maximizing the sampling period. The corresponding optimization problem is NP-complete. In order to cope with the exponential complexity of the optimization problem, this approach maps the problem onto Integer Linear Programming (ILP).

   Solving the corresponding ILP for large applications poses a stumbling block. Hence, we also developed polynomial-time algorithms that provide near-optimal solutions to the optimization problem. All heuristics are over-approximations and hence, sound (i.e., they do not overlook significant events). The first heuristic is a greedy algorithm that aims at adding state information to the auxiliary memory after the execution of significant events that participate in many execution branches. The second heuristic is based on a 2-approximation algorithm for solving the minimum vertex cover problem. The third heuristic uses genetic algorithms, where the population generation aims at minimizing the number of significant events that need to be stored in the auxiliary memory.

2. *Path-aware approach* [82]: This approach calculates the sampling period according to the execution path of the program. The static approach calculates the sampling period with respect to the complete control-flow graph of the program and disregards the actual path taken by the program at run time. Thus, the sampling period may be conservative with respect to the program's execution path. In other words, the monitor may take samples while a significant event has not executed since the last sample. To eliminate this unnecessary overhead, the path-aware approach uses symbolic execution to predict the execution path(s) of the program with respect to

the given input. Hence, the path-aware approach only considers the portion of the control-flow graph that is covered by the predicted execution path(s) when calculating the sampling period. As a result, the path-aware approach calculates a sampling period that is optimal with respect to the execution path. Further on, by merging the history approach with the path-aware approach, we amplify the reduction in the runtime overhead of time-triggered monitoring.

The history and path-aware approaches provide the means to establish an efficient time-triggered monitor for embedded systems running of single core machines. On the other hand, in the recent years, multi-core architectures have become common on embedded systems. To this end, we established an advanced time-triggered monitor that optimally monitors component-based real-time embedded systems running on multi-core architectures. Our studies show that the overhead imposed by the monitor in such systems is tightly coupled with two factors: (1) the sampling period of each time-triggered monitor running on each core, and (2) the mapping of components to computing cores. Our studies show that the overhead imposed by each time-triggered monitor not only depends on its sampling period, but also the structure of the components it monitors at run time. Finding the setting which results in the optimal overhead is NP-hard. In order to cope with the exponential complexity of the optimization problem, we map the problem onto ILP. The optimal solution provides the sampling period for each time-triggered monitor and a mapping of components to computing cores.

With respect to the promising outcome from time-triggered monitors, we gathered our work into a fully automated open source tool named RiTHM. The tool RiTHM takes a C program under inspection and a set of LTL properties as input and generates an instrumented C program that is verified at run time by a time-triggered monitor. RiTHM provides two techniques based on static analysis and control theory to minimize instrumentation of the input C program and monitoring intervention. The monitor's verification decision procedure is sound and complete and exploits the GPU many-core technology to speedup and encapsulate monitoring tasks.

Organization. Chapter 2 presents an overview of the most notable work on runtime verification. Chapter 3 presents an overview of runtime verification and event-triggered monitoring. Chapter 4 presents the formalization of the time-triggered monitor, along with the static approach for calculating the longest sampling period. Chapter 5 presents the history approach for reducing the monitoring overhead. Chapter 6 presents the path-aware approach for reducing the monitoring overhead. Chapter 7 presents time-triggered monitoring for component-based multi-core embedded systems. Chapter 8 presents the tool

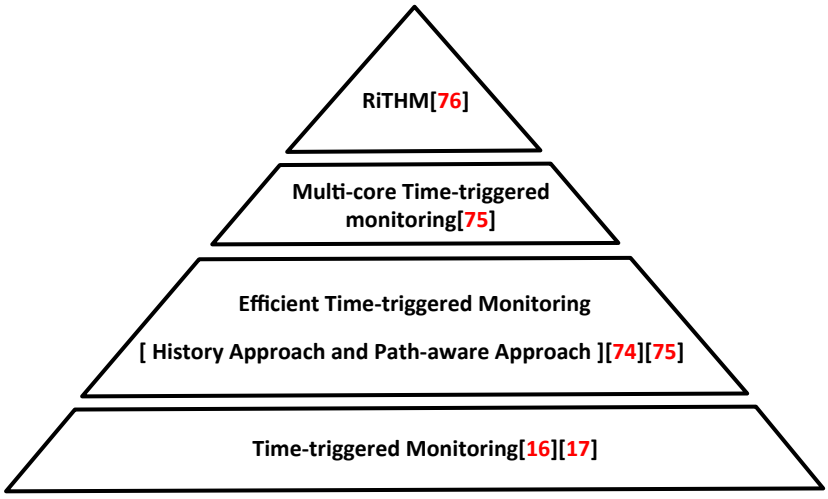


Figure 1.1: Contributions of this dissertation

RiTHM. Chapter 9 presents the conclusion and possible future work for runtime verification of real-time embedded systems.

1.1 Contributions

This dissertation provides a novel approach for runtime verification of real-time embedded systems. To our knowledge, there are not many published work which contribute to establishing runtime verification techniques that are suitable for real-time embedded systems. Figure 1.1 presents the four main contributions made within this dissertation.

- Creating a runtime verification technique suitable for real-time embedded systems. To this end, we established time-triggered monitoring which provides bounded overhead, predictable monitoring, and reduces over provisioning [18] [19].
- Creating an efficient time-triggered monitor. To this end, we established two approaches, history and path-aware, to reduce the number of samples taken by the monitor while still providing sound runtime verification [85] [82].
- Providing efficient time-triggered monitoring for embedded systems running on multi-core architecture. To this end, we established an optimization technique which calculates the setting (i.e., sampling period and mapping of components to cores) which results in the optimal runtime overhead imposed by the time-triggered monitors [83].

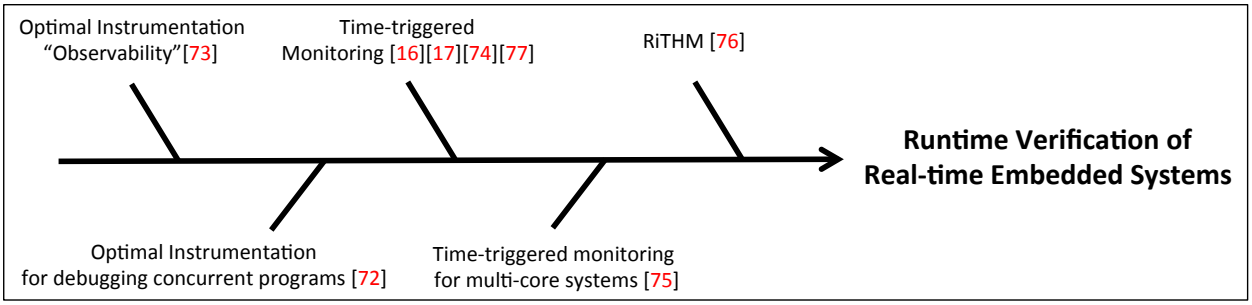


Figure 1.2: Work accomplished for runtime verification of real-time embedded systems

- A fully automated open source tool that provides time-triggered runtime verification for non-professional uses to verify their real-time embedded systems at run time [84].

In addition to the work presented in this dissertation, Figure 1.2 shows the work we have carried out to advance the opportunity to runtime verify real-time embedded systems [81] [80].

Chapter 2

Overview

In this chapter, we present work both on runtime monitoring and runtime verification which are relevant to our scope of research. We have come to realize that there are numerous works in the area of runtime monitoring which address issues similar to runtime verification. Hence, we believe they should not be excluded because of differences in terminology. To simplify, in the remaining of this proposal, we refer to both runtime monitoring and runtime verification as runtime verification.

In [32], the authors categorize runtime verification frameworks based on the following parameters:

1. *Specification Language:* It classifies the runtime verification frameworks based on the language they use to define properties. It considers the following parameters:
   (a) Language Type: The language used to specify a property can be based on algebra, automata, logic, etc..
   (b) Abstraction Level: It refers to the support that the language provides for specifying properties. It can be *domain-based*, *design-based*, or *implementation-based*. Languages which only support specification of properties for a specific domain are domain-based. Languages which support specification of properties related to the design of a system are design-based, and languages which support specification of properties related to the implementation of a system are implementation-based.
   (c) Monitoring Directives: It refers to the level which the properties must be verified. The levels are: program, module, statement, and event. Program-level

specify properties on threads or on relations between threads. Module-level specify properties on functions, procedures, methods, or components. Statement-level specify properties for a particular statement. Event-level specify properties on state changes or sequence of state changes.

2. *Monitor:* It classifies the runtime verification frameworks based on the implementation specifications of their monitor. It considers the following implementation parameters:

   (a) Monitoring Points: It refers to the points in the program where instrumentation is inserted. Monitoring points are inserted *manually* or *automatically*. Manual insertion manually finds the monitoring points and insert instrumentation into the program. Automatic insertion automatically detects monitoring points and inserts instrumentation. They use static, dynamic analysis or both to find the monitoring points.

   (b) Placement: It refers to where the monitoring code executes. The monitoring code can execute either *inline* or *offline*. Inline monitoring code is embedded in the program. On the other hand, offline monitoring code runs as a separate thread or a process. In the offline mode, when the program continues its run before the analysis of the monitor is complete, the monitor is asynchronous. In the other case, when the program halts its run until the monitor's analysis is complete, then the monitor is synchronous.

   (c) Platform: It refers to the monitor being implemented at the software level or the hardware level.

   (d) Implementation: It refers to how the monitor executes in relation to the original program. The different modes of execution are: *single-process*, *multiprogramming*, and *multiprocessor*. In the single process mode, the monitor executes in the same process as the original program. In the multiprogramming mode, the monitor and original program execute as separate processes or threads on the same processor. In the multiprocessor mode, the monitor and original program execute on different processors.

3. *Event Handler:* It classifies the runtime verification frameworks on how their monitor reacts when a property is violated or validated. The possible classes of reactions are as follows:

   (a) Control Level: It separates monitors that have the same reaction to any violation/validation from those that allow the user to specify separate reactions to different violations/validations.

(b) Response Effect: It refers to the different ways the monitors reaction can affect the program's behaviour. The monitor can have *no effect* on program behaviour and only report violations/validation. On the other hand, the monitor can also carry out graceful degradation, termination, or any recovery actions.

4. *Operational Issues:* It classifies the runtime verification frameworks based on the framework's external environment. The concerns surrounding the external environment are as follows:

   (a) Source Program Type: It considers the type of programs which the framework can monitor. The programs can be general-purpose, domain-specific, or category-specific.

   (b) Dependencies: It considers the restrictions on the frameworks. For instance, some framework can only operate on certain hardware, operating systems, or programs written in a specific language.

   (c) Level of Maturity: It rates the stage of development of the runtime verification framework.

In the remaining of this chapter, we use the aforementioned categories to present the work on runtime verification.

We present the work on runtime verification in three Sections:

1. *Monitor Synthesis:* This set of work focuses on developing tools for efficient synthesis of the monitor. They typically focus on developing efficient formal specification interpreters which can create the optimal finite state machine of the monitor. In addition, they tend to synthesis monitors which impose low overhead at runtime.

2. *Data Extraction:* This set of work focuses on reducing the overhead of the monitor when extracting data from the program run, to evaluate the properties. They typically focus on reducing the instrumentation instructions (i.e., instructions which send data from the program to the monitor). All the monitors presented in this chapter use an event-based method to extract the data. Meaning that the monitor extracts data from the program run when an event of interest (i.e., events that may change the evaluation of the properties) executes in the program.

3. *Runtime Verification Tools*: This set of work focuses on developing a tool that has both an efficient monitor synthesizer and framework for efficient data extraction.

2.1 Monitor Synthesis

To synthesize monitors, there are works which try to redefine properties in such a way that the synthesized monitor requires the least information regarding the execution trace to evaluate the properties. Havelund et al take such an approach in [50]. The authors present two algorithms for synthesizing an efficient monitor for safety properties written in past time LTL (ptLTL). Both algorithms depend on recursive definition of ptLTL operators. The authors prove that all ptLTL operators can be defined recursively. Hence, their synthesizer only requires access to the current and immediate previous program state. The algorithms are as follows:

1. *Formal Rewriting:* They rewrite ptLTL formulae by using Maude over Java PathExplorer (JPAX) [51]. They add rewriting rules regarding recursive definition of ptLTL operators to Maude. They base their rewriting rules on Hsiang's rewriting method to carry out efficient evaluation of the propositions in the ptLTL formula. In their model, JPAX runs Maude by loading their rewriting rules and the set of ptLTL properties into Maude. At every event generated by an instrumentation, Maude updates the value of each proposition in the ptLTL formulae using the recursive rules. Total complexity is $O(n \times m)$ where n is the number of events and m is the number of propositions to evaluate.

2. *Synthesizing a Monitor from LTL Formulae:* In this method, they use dynamic programming and the recursive rules to create a monitor. It generates the monitor either offline or inline. Their method enumerates all the sub-formulae of a ptLTL formulae and dynamically creates two matrixes where each entries of the matrix is a sub-formulae. At runtime, it updates the entries of the matrix using the recursive rules. One matrix shows the value of sub-formulae in the previous state and the other the value of the sub-formulae in the current state. The complexity is $O(n)$ for an event trace of size n.

This work resides in the category of logic-based languages with respect to its language type, and it resides in the category of implementation-based languages with respect to its abstraction level.

Some work tends to simplify monitor synthesis by providing frameworks which support multiple specification languages. In [7], the authors present a rule-based framework named *Eagle* for defining and implementing finite trace monitoring logics, including future and past LTL, extended regular expressions, real-time logics, interval logics, and forms of quantified temporal logics. In Eagle, monitoring carries out a state-by-state analysis, without

storing the execution trace. Eagle is influenced by MetateM where an LTL property is separated into pure past, present and pure future time formulas. The past formulas put a condition on the present-time formulas to determine how the state for the current point in time is built, and the pure future time formulas represent restrictions that need to be satisfied at some time in the future. As in [50], it uses recursive equations to evaluate temporal operators. In addition, an Eagle specification consists of a declaration section (D) and an observer section (O). D consists of zero or more rule definitions such as the recursive rewriting rules, and O consists of zero or more synthesized monitors which are in fact the properties of interest. The rules in D can be parameterized by formulas and primitive types. In Eagle, when an event of interest executes, a formula specification F (i.e., synthesized monitor) in O is updated based on the changes in the program state. Using the new program state and the rules defined in the declaration section, F evolves into a new formula F' which is monitored instead of F in the remaining of the program execution. The Eagle logical system shows to be expressively rich; any LTL, whose temporal modalities can be recursively defined over the next and past modalities [29], can be embedded within it. In addition, Eagle is a general-based language tool with respect to its specification language and abstraction level.

2.2 Data Extraction

To reduce the overhead of monitoring throughout a program run, researchers commonly focus on reducing the overhead of data extraction. One solution to this problem is by reducing the number of instrumentations required to monitor the program. Bodden et al in [15] , use a three-staged static analysis method to reduce the number of instrumentations where the properties are specified using tracematches [1]. Their analysis determines which events of interest (i.e., events related to the properties) needn't be monitored while still enabling the verification of tracematches. The analysis stages are as follows:

1. *Quick check:* It first instruments the program based on the tracematches (i.e., instruments events of interest) and counts the number of instrumentations per symbol of the tracematches. Second, it finds the set tracematches which incorporate symbols without any instrumentation. In the end, it deletes all the instrumentations which are only associated with this set of tracematches.

[1] *Tracematches* are an extension to AspectJ which allow programmers to specify traces via regular expressions of symbols with free variables

2. *Demand-driven points-to analysis* It uses a flow-insensitive analysis that finds inconsistent variable bindings. It first finds groups of instrumentations that have consistent variable binding for a tracematch. To do so, it extracts all paths leading to the match of a tracematch. Second, for each path, it extracts the groups of instrumentations with consistent variable bindings called *consistent shadow groups* that lead to the matching of the tracematch. In the end, it deletes the instrumentations that do not reside in consistent shadow groups.

3. *Flow-sensitive analysis:* It takes into account order of events of interest. As input, it takes the consistent shadow groups and the program call graph. Then, using the call graph, it extracts the order of execution of the instrumentations. Based on the execution order of the instrumentations, it finds the tracematches that never match regarding the program executions. Hence, it deletes instrumentations associated with such tracematches.

Their experimental results show that although the third stage carries out the most expensive analysis it does not notably reduce the instrumentation points. In other words, the first two stages already delete all the unnecessary instrumentations whose framework aims to find.

Based on [15], Bodden et al create a framework called *Clara* [17]. In addition to [15], Clara also presents *Dependency State Machines*. The idea behind the Dependency State Machines is that some events of interest are only of importance if they are executed before or after another set of events of interest. In this case, after Clara executes the analysis stages from [15], using the Dependency State Machines, it finds additional instrumentations which are meaningless without the deleted instrumentations. Hence, it also deletes these additional instrumentations.

In the same line of work, Bodden et al [16] try to improve their work in [15] by focusing on developing an ahead-of-time technique that finds program locations with absence of property violations. Hence, they only instrument program locations with violation probability. To do so, they carry out a flow-sensitive analysis after running the first two stage analysis from [15]. This new flow sensitive analysis stage, is at a per-method and per-tracematch level. The analysis takes the following steps: For each method m the analysis determines the set of configurations each tracematch can have when m starts execution. Then, it checks if the statements of m can evolve the configuration such that the tracematch matches. If so, it preserves all the instrumentations contributing to the tracematch. In addition, their method incorporates a machine learning tool which accumulates data from the program run. Using the accumulated data, the learning mechanism deletes instrumentation points when they do not provide additional information for checking tracematches.

Based on the idea from [16] (i.e., only monitor portions of the code which may violate a property), Kim et al [57] present a monitoring technique for Software Product Lines (SPL). Their method finds the feature combinations (i.e. programs) that never violate the properties. To do so, they advance Clara [17] to understand feature combinations. First, for each tracematch, their method finds all the symbols that need to be executed for the tracematch to match. Then, their method does not add instrumentation for the tracematch to a feature combination, if none of the execution paths of the combination incorporate all of the required symbols for the tracematch. Note that a symbol is present if any of the instrumentations associated with the symbol executes in the path. Meaning, if there exists an execution condition which an instrumentation from each symbol executes, the feature combination must be instrumented. Their analysis starts from a sound but conservative instrumentation, then step by step refines the instrumentation as it goes through the program's call graph.

Another method to reduce the overhead of monitoring, is the distribution of the cost of data extraction among multiple users. Bodden et al [14] take this approach. In their method, they distribute instrumentations in two ways:

1. *Spatial:* It uses the first two analysis stages from [15] to find the consistent shadow groups. In this case, the instrumentation in the program copy of each user is the result of the union of one or more consistent shadow groups. In addition to these instrumentations, there are instrumentations that help in preventing false positives (called skip shadows). The problem with this method is that some users may have a set of instrumentations that induce large overhead.

2. *Temporal:* It turns the instrumentations on and off over time. This method keeps skip shadows on at all times except when a skip shadow's tracematch can never be matched. The problem with temporal partitioning is that the verification of any property can be ignored at runtime.

This method uses a centralized server that combines the data from the user set to conduct analysis on the program.

Similarly, to reduce monitoring overhead, the authors in [67] propose a remote sampling infrastructure to collect data on a program's run-time behaviour from a large user community and analyze the data to detect bugs. To debug a program, the authors present a remote sampling infrastructure which has the following steps: (1) *Sampling:* the program is instrumented based on the description of a bug. For the same execution path, each program run by a user only executes a subset of the instrumentations. Hence, different program copies instrument different data along the same execution path, (2) *Data Collection:*

the data collected by the instrumentation is sent to a central site over the network, and (3) *Analysis:* The central site analyses the data and narrows down the possible locations of both deterministic and non-deterministic bugs. In the *sampling step*, the framework adds instrumentation instructions based on a bug description. In this work, each instrumentation stores the value of a predicate which tests a specific program state. To reduce the instrumentation overhead in the program, each instrumentation is executed based on a specific probability. If the instrumentations of a portion of a code (i.e., basic block) are improbable to be executed, the execution is redirected to the un-instrumented version of the basic block. The authors use geometric distribution to predict if the instrumentations will execute or not. In the *data collection step*, the framework sends a vector incorporating the value of predicates stored by the executed instrumentations to a central site. The site groups data regarding the same execution path of program runs from different users. In the *analysis step*, to find deterministic bugs, the framework eliminates the predicates which do not affect a faulty run. Meaning, they only keep track of predicates which their value is different from when the run is successful. These predicates help the developer in narrowing down the search to portions of the code which affect the program states tested by these predicates. In addition, the authors use statistical analyses to find codes possibly causing nondeterministic bugs. They rank the predicates based on the possibility that their tested state is causing the bug. They use machine learning and *logistic regression* to create a binary classifier to rank the predicates.

Some tend to reduce the overhead of data extraction by inserting instrumentation on the fly as the program runs. Brorkens et al in [22] take such an approach. [22] presents a framework named *jassda* which inserts instrumentations on the fly to Java programs based on CSP-like property specifications. To reduce changes to the source code, jassda uses the Java Debug Interface (JDI) from the Java Platform Debugger Architecture (JPDA) to allow the Java virtual machine to emit the events from the execution of instrumentation. Jassda has two stages:

1. *Initial stage:* Jassda first connects to the running program via JDI. Second, it statically analyzes the loaded classes to find the set of events that might be generated during their execution. Then, according the the CSP properties, jassda instruments the events of interest using JDI.

2. *Runtime stage:* Jassda evaluates the CSP properties when an event of interest executes. In addition, jassda instruments the events which dynamically load classes at runtime. Hence, when a new class is loaded, jassda configures the set of instrumentations regarding events of the newly loaded class.

In addition, jassda also allows additional specification languages to be added.

In [33] [92], Dwyer et al tend to choose a sub-set of properties based on the limitations on tolerable monitoring overhead to monitor. They present an approach to compose properties to form a single integrated property that is then systematically decomposed into a lattice of properties. This lattice encodes the conditions which need to be satisfied for property violations. Using this lattice, they select a set of properties to monitor, such that they lower the monitoring cost to a value which can be tolerated by the system while preserving sound violation detection. They consider sound violation detection to be the possibility of failing to report a violation but never reporting a violation when one does not occur. They consider the monitoring overhead to be: (1) the number of instrumentations and (2) the cost of monitoring each property for each allocated instance of a class. Their method takes the following approach:

1. They show that one way to reduce monitoring overhead is to integrate small properties into larger properties. This integrated property has a richer space of sub-properties that vary in monitoring overhead.

2. They systematically decompose the integrated property into a lattice of properties where each has a different set of instrumented events of interest. These properties are less expensive to monitor since they require less instrumentation. Hence, to reduce the monitoring overhead, they choose a subset of the sub-properties of the integrated property to reduce the number of instrumentations.

3. They define a method for exploiting the structure of the lattice to choose the subset of sub-properties. In general, they provide a threshold on the monitoring overhead (T) as input to their method. Their method operates as follows: while the monitoring overhead is below T, it selects the best sub-property from the lattice and updates the set of properties being monitored (M). It then estimates the gap between the current monitoring overhead and T. It filters the remaining sub-properties by removing the ones that are subsumed by properties in M or impose high cost (with respect to the gap). Then, it chooses the next sub-property. They choose the "best" sub-property based on: potential value added with respect to the violation detection relative to a property set, subset of properties that are most different with respect to the number of unshared events of interest [33], or properties based on a given cost threshold.

They show that this leads to reductions in the monitoring overhead while retaining violation detection power.

To reduce monitoring overhead, in [34], Dwyer et al also present a method which adapts the instrumentation in the program according to the execution path. Their method, Adaptive Online Program Analysis (AOPA) is based on the idea that at any time during a verification of stateful properties only a subset of program behaviour should be monitored. AOPA removes and adds instrumentation to verify program properties *relevant* to the program state. AOPA guarantees to produce the same results where all instrumentations execute throughout the program run. They implement AOPA over the Sofya [62] framework. Users of AOPA request the observation of events using a specification. Sofya uses a combination of byte code instrumentation and the Java Debug Interface (JDI) [34]. The authors use the debug interface in Java 1.5 to add features in Sofya that enable the addition and removal of byte code instrumentation during execution. For AOPA to determine what instrumentation to remove or add, the authors map the properties to an finite state automaton (FSA). The intuition is that at each state in the FSA, they only instrument events of transitions which leave the given state, and the remaining of the events are overlooked in that state. Ignoring such symbols in a trace does not change the acceptance of the trace by an FSA. In this setting, when an event is observed, AOPA updates the current state based on the FSA, and adds/deletes instrumentation according to the new state. This technique ensures that all instrumentation are enabled for transitioning out of the next state, which ensures equivalence with online FSA checking. AOPA is also capable of checking FSA conformance of independent objects during program execution as well.

There is a line of work which are not concerned with runtime verification; instead they focus on dynamically reducing instrumentation overhead in testing. We believe their techniques can aid in reducing data extraction overhead in runtime verification. In [29], the authors tend to remove an instrumentation after it is executed. They note that in spite of the benefits of removing instrumentation to reduce instrumentation overhead, its applications is challenging because: (1) It requires for the target program to be stopped, re- built, and re-executed, (2) It cannot take advantage of multiple instances of the program that may run in parallel [29]. The authors present two techniques that address these challenges to dispose instrumentation of the program at runtime. They implemented both following techniques for the Java language by changing the Java Virtual Machine. Their first technique, *Local Disposal (LD)*, removes each instrumentation immediately after its execution. The removal takes two steps. First, LD detects the occurrence of execution of an instrumentation. Second, they modified the Kaffe JVM implementation so LD can remove the instrumentation from the bytecode and adjust the addressing and stack of the running program without stopping its run. More specifically, they modify the execution engine within the JVM. Their results show that test suites that generate repetitive execution patterns make the most advantage of LD. Their second technique, *collective disposal (CD)*, enhances

LD by considering the execution of instrumentation from multiple running instances of the program. Each program instance has a vector showing whether an instrumentation has been executed or not by any of the other program instances. If the vector shows that an instrumentation has not yet been executed, then the steps resemble to applying LD. If the vector indicates that an instrumentation has been executed by another instance, then the instrumentation is removed without ever being executed. They implemented CD using a centralized server which synchronizes the vector in each instance.

In [76], the authors present a framework to implement test coverage which uses a demand-driven approach based on the execution paths of the program to insert or remove instrumentations. This framework uses dynamic instrumentation on the binary code of the program that can be inserted or removed at run time to provide high performance and low memory overhead. The prototype of their framework is named Jazz. Jazz includes a test specifier, a test planner, a dynamic instrumentor, and a test analyzer. The framework includes a special language for specifying software test processes. A test planner consumes the specification and generates a test plan which shows the program locations where instrumentation should be added and when in the execution each instrumentation can be removed. There are three cases the planner has to consider when deciding when to insert or delete instrumentation. First, it must find the set of instrumentations that are needed initially in the program. Second, it needs to determine which instrumentations can be inserted and removed on-demand along the execution path of the program at runtime. Finally, the planner has to determine the lifetime of an instrumentation and whether it must be re-inserted after being removed [76]. The plan is stored in a probe location table (PLT). A PLT entry has an instrumentation type, a payload, and a list of instrumentations to insert and remove. Using the generated plan, the dynamic instrumentor inserts or removes the program instrumentation at run time to carry out the specified tests [76]. In the end, the framework uses a test analyzer to report results. The instrumentor executes according to the path that the program takes during execution. When the program reaches an instrumentation, the instrumentor uses the test plan for inserting and deleting instrumentation. Jazz is incorporated in Eclipse and Jikes for the Intel x86.

The authors in [103] also present an approach to dynamically insert and remove instrumentation in C code to reduce the runtime overhead of code coverage. They use the information from the dominator tree of the program to reduce the initial number of instrumentations needed in the program. Using the dominator tree they infer the coverage of a basic block by coverage of other basic block(s). Hence, they reduce the number of instrumentation by only instrumenting the leaf basic blocks of the dominator tree of a function while increasing the coverage information obtained per instrumentation. They also instrument a basic block if it has at least one outgoing edge to a basic block that does

not dominate. Hence, using the information from the dominator trees, they locate the instrumentations which need to be inserted at runtime. Then, they use incremental instrumentation to insert the necessary instrumentation code when a function is called for the first time during program execution [103]. To do so, before running the program, they only insert breakpoints at the beginning of each function. During the execution of the program, when a breakpoint is reached, the dominator tree of that function is generated and the necessary instrumentation code is inserted. They automatically delete the instrumentation just after the first time it is executed. To dynamically insert and delete instrumentation at runtime, they develop their framework over Dyninst [23]. Their experiments show that their approach reduces runtime overhead by 38-90%. They note their work can be extended to other dynamic instrumentation tools such as Dtrace [26], DIOTA [71], and Valgrind [87].

A work which is quite similar to our path-aware approach is [99]. The authors use state estimation of Hidden Markov Model (HMM) to estimate the probability that a run of a program satisfies the set of temporal properties when they reduce monitoring overhead by using sampling. In this paper, the authors see the observed event sequences as observation sequences of a Hidden Markov Model (HMM). Then, they use the HMM model of the program and the classic forward algorithm for HMM state estimation to estimate the program states lost in the gap between samples [99]. Using the estimated states, they compute the probability that the run of the program property satisfies the properties. They automatically extract the HMM for a system by using complete traces extracted from the standard HMM learning algorithms. They note that the created HMM is not required to be the exact model of the system. They keep track of the state of the synthesized monitor by advancing the forward algorithm. They add an auxiliary function to calculate the probability pi of transitions of the monitor during gaps between the samples. Meaning that when HMM is in state s and the Monitor is in state m, pi(m,n) is the probability that the next observation (i.e., the observation in the state after s) causes the Monitor to transition to state n [99]. They also consider another auxiliary function g, called the gap transition relation, which they use to compute the overall effect of a gap with a specific length l. g(s,m,s',n) is the probability that, if HMM is in state s, the Monitor is in state m, and there is a gap with a specific length, then the HMM is in state s and the Monitor is in state n after the gap with a certain probability [99]. In addition, their approach supports parameterized temporal properties [99].

In [54] [53], the authors introduce the technique of Software Monitoring with Controllable Overhead (SMCO), which is based on a combination of supervisory control theory of discrete event systems and PID-control theory of discrete time systems. SMCO controls monitoring overhead by temporarily disabling monitoring of selected events for as short a time as possible under the constraint of a user-supplied monitoring overhead head o_t.

They claim their technique is optimal in the sense that it allows SMCO to monitor as many events as possible while keeping the monitoring overhead below o_t. In other words, given the o_t, the SMCO controller periodically sends enable/disable monitoring commands to an instrumented program and its monitor such that the monitoring overhead stays below o_t at all times. The SMCO controller is a feed-back controller; i .e., the current overhead level is continually fed back. This enables the controller to monitor the overhead and, as a result, disable monitoring when the overhead is close to o_t and, conversely, enable monitoring when there is a small possibility that the overhead goes beyond o_t. In this setting, the plant P is a state machine, the desired outcome is a language L, and the controller design problem is that of designing a controller Q, which is also a state machine [54] [53], such that the language $L(Q||P)$ of the composition of Q and P is included in L. In addition, they consider two types of PID controllers: (1) a single, integral-like global controller for all monitored objects in the software; and (2) a cascade controller, consisting of an integral-like primary controller that is made up of a number of proportional-like secondary controllers, one for each monitored object in the application [54].

2.3 Runtime Verification Tools

Most runtime verification tools focus on monitoring Java programs. One of such tools is Java-MoP [28]. Java-MoP is a specialized version of the Monitoring-oriented Programming (MoP) [27]. MoP is an approach which enables the synthesis of the monitor from a formal specification. At the pre-compilation stage, MoP takes formal specifications and creates monitoring code (i.e., code representing the monitor) in the same language of the program. In this stage, MoP finds the events of interest that must be tracked for monitoring and adds appropriate monitoring code using AOP. MoP has four layers which enables the separation of the generation and integration of the monitor. It has standardized protocols between the layers which enables MoP to be extended to new specification languages and programming languages. Its highest layer is the interface layer, then a specification processors layer that handles the integration of monitoring code. Each processor has a scanner and a transformer which is specialized for a programming language. The scanner extracts the formal specifications and sends them to the lower layer. The transformer gets the monitoring code from the lower layers and integrates them into the code. The monitor is inline, offline synchronous, or offline asynchronous. The last two layers are the language shell and the logic engine. The language shell generates programming language and specification formalism specific to the monitoring code. The logic engine synthesizes monitors from the specifications. In addition, MoP already supports formal specification

for past time and future time LTL as well as extended regular expressions. MoP also handles violations by executing recovery code, displaying or sending messages, or throwing exceptions.

Java-MOP has a distributed client-server architecture. The client provides both a textual and a graphical user interface which takes one or more names of annotated Java files as input. As output, it generates corresponding files in which monitors are synthesized and integrated appropriately.

Another monitoring framework specialized for Java programs is Java-Mac [59]. Java-MaC is a prototype implementation of the Monitoring and Checking (MaC) architecture for Java programs. MaC analyses the temporal and data behaviour of a program run based on the program's executed events and their execution conditions. The MaC architecture is open to any programming language. Mac represents the properties at both high level and low level. The high level representation is in the Meta Event Definition Language (MEDL). Events in the properties presented in MEDL needn't precisely map to the events in the program. The mapping from high level events to events in the program is done by the low level representation in the Primitive Event Definition Language (PEDL). MaC has a static and run-time phase.The static phase automatically creates run-time components: the filter, event recognizer, and run-time checker. A filter is a set of instrumentations which keep track of changes of monitored objects and send changes in the program state to the event recognizer. PEDL specifications define what information is sent from the filter to the event recognizer [59]. PEDL specifications note how information from the filter is transformed into events in MEDL specifications. Using PEDL, the event recognizer recognizes events in the information from the filter which map to events in the MEDL specifications. In the next step, the event recognizer sends the recognized events to the run-time checker. The run-time checker checks if the execution event history satisfies the MEDL specifications. The run-time phase extracts information from the running program and verifies the specifications given in MEDL.

As for Java-Mac [59], it has three components: an instrumentor, a PEDL compiler, and a MEDL compiler. The PEDL compiler compiles a PEDL specification into an abstract syntax tree (AST) used by the event recognizer. Also, a PEDL compiler generates a list of variables/methods of interest (instrumentation information) that is used by the instrumentor. The instrumentor takes a Java bytecode and the instrumentation information to insert a filter into the bytecode. The filter has a communication channel, set of instrumentations, and a filter thread. The communication channel is from the program to the event recognizer used to send extracted information from the instrumentations. The MEDL compiler also compiles a MEDL specification into an AST which is used by the run-time checker. In addition, Java-MaC HAS three different communication mechanisms: TCP socket commu-

nication, communication through a FIFO file, and communication channel implemented by a user using InputStream and OutputStream provided by Java-MaC API [59].

In [94], the authors advance MaC to monitor real-time programs. They focus on the verification of timeliness and reliability correctness by enabling MaC to process and monitor quantitative and probabilistic property specifications. They refer to it as RT-MaC. They add time-bound temporal operators and probabilistic operators to achieve quantitative and probabilistic property specification. For time-bound temporal properties, they advance MEDL to incorporate time. They refer to it as RT-MEDL and it is as expressive as MTL. For probabilistic properties, when verifying probabilistic properties, they use statistical analysis to calculate confidence intervals. Since the statistical analysis has access to one execution path, the only way that they were able to collect multiple samples from one execution path was to use a system with repeating or periodic behaviours such as soft real-time schedulers, network protocols or web servers [94]. Each repetitive behaviour is used as a sample execution (i.e., experiment). With a set of experiments and a probabilistic property, RT-MaC statistically checks whether a system satisfies a property by using sequential hypothesis testing [94]. The sequential hypothesis testing depends on an outcome of previous analysis of the probabilistic property. After each experiment, the sequential hypothesis testing says the probabilistic property is either satisfied, not satisfied, or needs more experiments. In case the program run terminates early, RT-MaC says a probabilistic property is satisfied or not satisfied, while providing a quantitative measure of confidence in both case. For quantitative properties, upon receiving a new event from the instrumentation, the RT-MaC also evaluates the timeout of time-bound properties. A timeout signal initiates the evaluation and updates the values of these conditions as soon as their values change. The overhead depends on communication cost, and cost of running timers. The only overhead that RT-MaC adds to the communication cost is the deadline messages exchanged between the runtime checker and the filter [94].

Another framework to monitor Java programs is Java PathExplorer (JPAX) [51]. JPAX monitors the execution of a Java program against specifications formulated in past time or future time LTL. Using JPAX, the users write temporal specifications in the Maude rewriting logic as presented in [50]. As for future time LTL, JPAX generates a special finite state machine (FSM), called binary transition tree finite state machine (BTT-FSM) that is actually the monitor. In the BTT-FSM, each state represents a BTT of propositions which need to be evaluated. Based on the result of the proposition, JPAX parses the BTT downwards so other propositions get evaluated as well. The leaves of the BTT represent states. Based on the leaf state which the proposition evaluation leads to, the BTT-FSM makes a transition. BTT-FSMs are required to only check at most all the propositions in the property in order to move onto the next state when it receives a new event. JPAX also

checks the program for concurrency errors such as deadlocks and data races. Using a single random execution trace, JPAX can conclude the existence of deadlocks and data races in other traces of the program as well. JPA X consists of an instrumentation module and an observer module. The instrumentation module uses the JTrek Java bytecode engineering tool module to automatically instrument the bytecode by adding new instrumentation instructions. The observer reads the events from the instrumentations and analyses the satisfaction of the properties regarding the executed events. In concurrency analysis, the events of interest to JPAX are the taking or releasing of locks and accesses to variables (needed for data race analysis).

One of the rare runtime verification tools which is geared towards C programs is RMOR [48]. In general, RMOR takes a C program with a textual RCAT specification presenting the properties. In return, using the RCAT specifications, RMOR develops a monitor and an instrumented C program. RCAT is an LTL inspired graphical state machine language editor which can express both safety and liveness properties. The RCAT specification also notes how the events in the specifications map to code fragments in the program which enables automated instrumentation. [48] implements RMOR over CIL [86]. CIL parses the C program and generates its AST with type information. RMOR uses the AST to (1) enable RCAT to turn specifications into FSMs, then into monitors, and (2) automatically instrument the program. The CIL parser takes the given FSMs and creates a monitor in a separate C file. In other words, the monitor developed by CIL's parser is a set of synthesized C programs modelling the FSMs created by RCAT. As for the automatic instrumentation, RMOR has an instrumentation module using the CIL visitation pattern framework. The instrumentation module scans the AST and inserts instrumentation accordingly. Each instrumentation point indicates an event in the FSMs. When the instrumentation of an event executes, the instrumentation module notifies the monitor that an event has happened. In addition, RMOR can create two types of reactions to a violation, either a simple standard output or can define an error handler for different violations.

InterAspect [96] is an framework instrumentation framework over GCC which also focuses on instrumentation of C programs to aid runtime verification. InterAspect enables instrumentation plug-ins to be developed using AOP pointcuts, join points, and advice functions. InterAspect also supports customized instrumentation based on results from static analysis. InterAspect is implemented using the GCC plug-in API for manipulating GIMPLE. It hides the complexity of GCC's API while it presents a new aspect-oriented API in which instrumentation is conducted by defining pointcuts. InterAspect makes use of a specification compiler. The specification compiler cuts down an AOP specification into pointcut definitions, associated weaving instructions, and advice code. The first two

aspects are sent to an InterAspect-based weave module to be evaluated while the instrumentation plug-in pass is being executed, on the other hand, the advice code is sent to GCC to be compiled [96]. InterAspect runs an instrumentation plug-in pass that traverses the GIMPLE code to find program points that are join points in a specified pointcut. Then, for each such join point, it calls a user-provided weaving callback function, which can insert calls to advice functions [96]. Callback functions have access to information about each join point. Hence, InterAspect uses this information to enable customization of the inserted instrumentation. In addition, InterAspect enables the possibility to toggle instrumentation at the function level by duplicating functions. InterAspect creates two copies of a function, an instrumented and non-instrumented version. When joining on a pointcut, the caller specifies the copy that should take part in the join. Hence, InterAspect enables the possibility to turn on and off the instrumentation of a function.

In [40] [41], the authors present the Artemis framework which is a compiler-based instrumentation framework. Artemis guides monitoring techniques toward locations of the program where bugs are likely to occur while imposing low monitoring overhead. Artemis also provides system-load aware runtime monitoring that allows the monitoring coverage to be dynamically scaled up to take advantage of extra CPU cycles when the system load is low [40] [41], and dynamically scaled down to monitor only the most suspicious regions when the system load is high. Artemis can operate in three modes: (1) default mode, where it carries out baseline monitoring only when the executed context is new, (2) sampling mode, where it also carries out baseline monitoring with a predefined sampling rate even when the context is not new, and (2) full-monitoring mode, where it always carries out baseline monitoring regardless of whether or not the context is new [40] [41]. Artemis dynamically switches between the modes, and the sampling rate in the sampling mode can be dynamically adjusted. The current implementation is based on source level instrumentation using the Cetus C Compiler, and is applicable to any hardware/OS platform. Their technique is predicated on the idea that if the global context of a region (the state of all in-scope variables, method parameters, and storage reachable from these) is the same then the outcome of an execution of the region, when projected onto the outcomes of "correct" and "buggy", will tend, with a high probability, to be the same [40] [41]. The Artemis framework is used as follows. The program is automatically instrumented by the compiler with context checks and the monitoring required by the baseline monitoring technique. At runtime, when the program's context for a region does not belong to the context invariant for the region, the instrumented version of the region is chosen for execution, and the context invariant for the region is updated. Note that every time that the region executes, Artemis either chooses to execute the instrumented or non-instrumented version. In the sampling mode, Artemis chooses the instrumented version with a dynamically adjustable sampling

rate even when the current context belongs to the learned context invariant [40] [41]. In the full-monitoring mode, the instrumented version is always chosen. The Artemis framework yields a performance floor overhead of 5.57%.

Chapter 3

Runtime Verification

In Chapter 1, we presented numerous catastrophic outcomes of malfunctions in real-time embedded systems. These outcomes reflect the fact that testing and verification are not sufficient techniques to ensure correctness of such systems at run time. Although, both testing and verification have unique characteristics which are required to ensure correctness of the behavior of a system at run time. Testing has the ability to observe and trace the behavior of the system at run time. Verification provides a sound and complete verification engine that is capable of verifying the behavior of the system. As a result, *runtime verification* was established which is a technique where a monitor checks at run time whether or not the execution of a system under inspection satisfies a given correctness property. In other words, runtime verification combines testing's ability to observe the runtime behavior of the system with verification's sound and complete verification engine.

In this chapter, we thoroughly explain the building blocks of runtime verification in Section 3.1 and event-triggered monitoring which is the most common monitoring technique used in runtime verification frameworks in Section 3.2.

3.1 Building Blocks of Runtime Verification

Figure 3.1 shows the building blocks and data flow of a generic runtime verification framework. A runtime verification framework consists of the following main components.

- *Observer*: The observer extracts the state of the program to be verified against a set of correctness properties. The observer is invoked at various points of the program

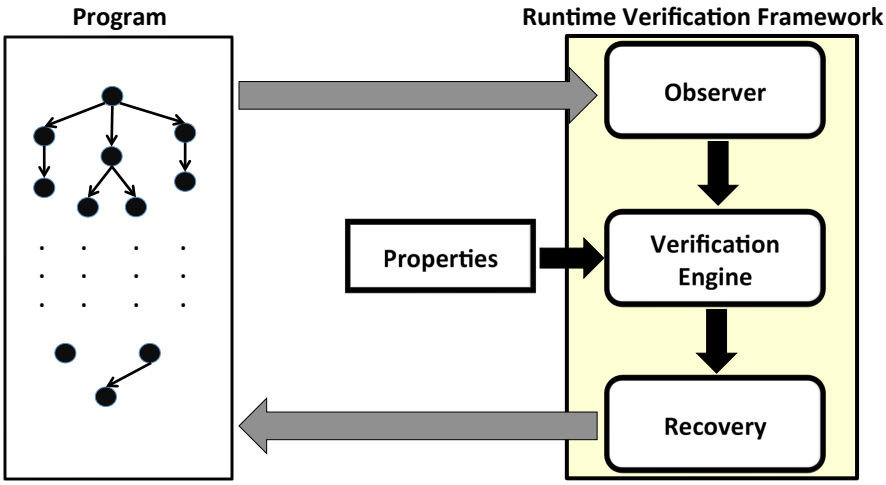


Figure 3.1: Building blocks of a generic runtime verification framework

run. The invocation type of the observer can differ from one runtime verification framework to another. As it will be discussed in Section 3.2, event-triggered monitoring is the common invocation technique among main stream runtime verification frameworks where after the execution of an instruction of interest, the program invokes the observer. An instruction of interest is an instruction that may change the evaluation of at least one of the properties. We elaborate more on these instructions in Chapter 4. After extracting the state of the program, the observer provides the data to the verification engine.

- *Verification Engine*: The input to the verification engine is the state of the program and a set of correctness properties. The verification engine takes the state of the program and evaluates the properties to check if the program has violated any of the properties. The verification technique used by the verification engine depends on the specification language of the input properties (ex. regular expression, LTL, etc.) and the formal representation of the program state (ex. finite state machine, automata, state transition system, etc.). In the case where a property is evaluated to False, the verification engine can either invoke a recovery module to handle the violated property or only log the state of the property. The verification engine is known to be referred to as the *monitor*.

- *Recovery (Optional)*: The runtime verification framework can have a recovery component which is invoked by the verification engine when a property is violated. The recovery component can devise a solution to redirect the execution of the program

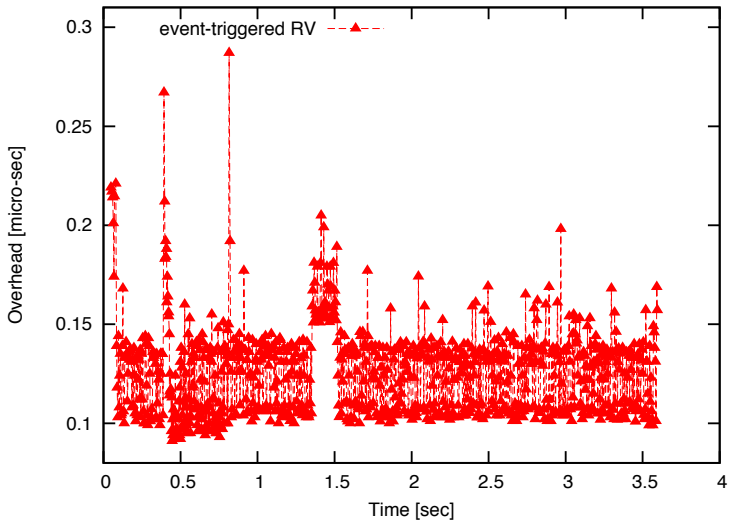


Figure 3.2: Event-triggered monitoring of blowfish

such that the program satisfies the property from here on after. To this end, the recovery module interferes with the execution of the program, for instance by changing register values, and changes the program's course of execution. In most cases, the recovery module terminates the execution of the program in the program's safe state.

In this thesis, our focus is on the design and analysis of the observer. As discussed in Section 3.2, the runtime behavior of the observer has a significant role in the runtime behavior and hence, the applicability of the runtime verification framework.

3.2 Event-Triggered Monitoring

The invocation technique of the observer component has a high impact on the runtime behavior of the runtime verification framework. The most common invocation technique has been *event-triggered*. In event-triggered invocation, the observer is invoked after the program under scrutiny executes an instruction of interest. To this end, the invocation of the observer is delegated to the program under scrutiny. A runtime verification framework which uses an event-triggered to invoke the observer and hence, the verification engine, is referred to as *event-triggered monitoring*.

Event-triggered monitoring has the following distinguishable runtime characteristics.

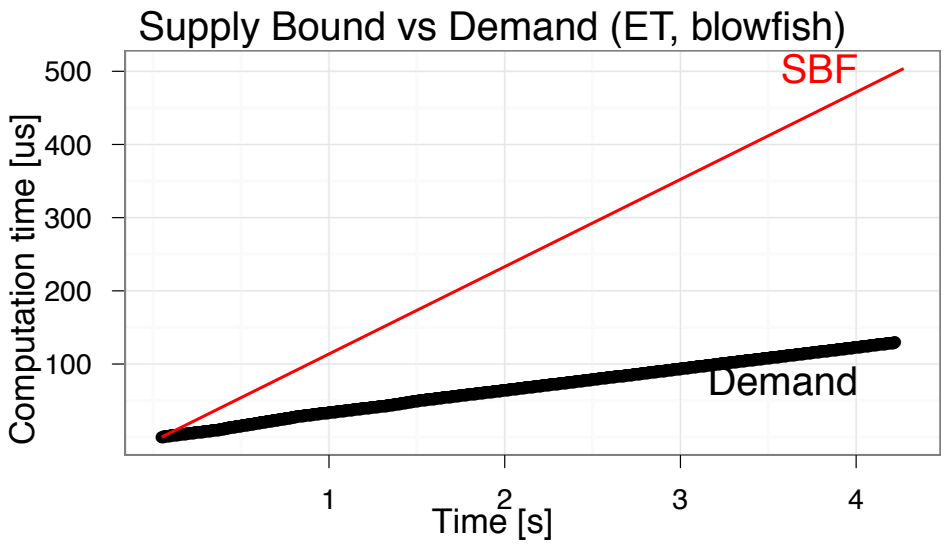


Figure 3.3: Supply bound and demand bound function for event-triggered monitoring of blowfish

- *Jittery Overhead*: At each observer invocation, the observer *only* extracts the program state changed by the instruction of interest which caused the observer invocation. For instance, if the instruction of interest changed the value of variable x, then the observer will only extract the new value of variable x. As a result, the overhead imposed by the observer from one invocation to another is tightly coupled with the type of variable extracted by the observer. To this end, the overhead imposed by the observer can severely differ from one invocation to another. Figure 3.2 shows the event-triggered monitoring of program blowfish from the MiBench [1] bench suite. Each data point in Figure 3.2 represents the absolute overhead imposed by invoking the observer;i.e., the overhead of halting the program execution, calling the observer, extraction of the program state by the observer, and resuming of the program execution. As can be seen, event-triggered monitoring imposes jittery overhead throughout the program run.

- *Unpredictable Invocations*: Since the observer is invoked after the execution of an instruction of interest, the invocations are unevenly distributed throughout the program run. To this end, there can be portions of the program execution with high density of instructions of interest which leads to a high volume of observer invocations. In Figure 3.2, it can be seen that there is a high density of observer invocations from 0.5 to 0.7 seconds of the program run.

- *Over-provisioning of Resources*: The jittery overhead of observer invocations and their uneven distribution throughout the program run causes over-provisioning of resources. Figure 3.2 shows the supply bound function and demand bound function required to carry out the event-triggered monitoring of blowfish. The supply bound function $sbf_{\Gamma}(t)$ is the minimum resource supply by resource Γ (i.e., the processor is considered the resource in this thesis) over all intervals of length t throughout the program run. The resource demand bound function $dbf(W, At)$ is the maximum possible resource demand of a task set W under algorithm A during an interval of length t. Over-provisioning of resources can be seen by the large gap between the supply bound function and demand bound function. For a detailed look into the underlying mathematics of the supply and demand bound function refer to [36] [66].

These characteristics of event-triggered monitoring can lead to transient over load situations in real-time embedded systems. As a result, it can result in catastrophic outcomes in safety-critical and mission-critical systems.

Chapter 4

Time Triggered Monitoring

As mentioned in Chapter 3, event-triggered monitors may cause transient overload situation in real-time embedded systems. Hence, event-triggered invocation is unsuitable for runtime verification of such systems. Our studies show that the technique used to invoke the observer for real-time embedded systems must have the following characteristics.

- *Bounded Overhead*: The deviance between the absolute overhead from one invocation of the observer (i.e., monitoring sample) to another must be minimal. In other words, the monitoring samples must not have jittery overhead.
- *Efficient Monitoring:* The overall overhead imposed by the monitoring samples must not saturate the resources of the system.
- *Reduced Over-provisioning:* The gap between the supply bound function and the demand bound function must be minimized.

The candidate invocation scheme that satisfies the above characteristics has been *time-triggered* monitoring where the observer and hence, the verification engine, is invoked based on a predefined time interval. In this chapter, we thoroughly elaborate on our time-triggered monitor [21] [19].

4.1 Preliminaries

In this section, we present the preliminary concepts. In Subsection 4.1.1, we present the notion of *control-flow graphs* for analyzing timing characteristics of programs written in

high-level programming languages. In Subsections 4.1.2 and 4.1.3, we present the concept of timed automata [4] and 3-valued linear temporal logic, respectively, as basis for presenting the semantics of time-triggered runtime verification in Section 4.2.

4.1.1 Control-flow Graph

Definition 1 (Control-flow Graph) *The* control-flow graph *of a program P is a weighted directed simple graph $CFG_P = \langle V, v^0, A, w \rangle$, where:*

- *V: is a set of* vertices*, each representing a basic block of P. Each basic block consists of a sequence of instructions in P.*
- *v^0: is the* initial vertex *with indegree 0, which represents the initial basic block of P.*
- *A: is a set of* arcs *(u, v), where $u, v \in V$. An arc (u, v) exists in A, if and only if the execution of basic block u can immediately lead to the execution of basic block v.*
- *w: is a function $w : A \to \mathbb{N}$, which defines a* weight *for each arc in A. The weight of an arc is the* best-case execution time *(BCET) of the source basic block*[1]. □

Notation: Let v be a vertex of a control-flow graph. Since the weight of all outgoing arcs from v are equal, we denote the weight of the arcs that originate from v by $w(v)$.

For example, consider the C program in Figure 4.1(a). If each instruction takes one time unit to execute, the resulting control-flow graph is shown in Figure 4.1(b). Vertices of the graph in Figure 4.1(b) are annotated by the corresponding line numbers of the C program in Figure 4.1(a).

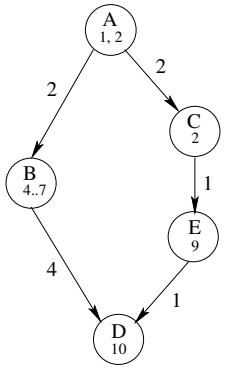
4.1.2 Timed Automata

Let AP be a finite set of *atomic propositions* and $\Sigma = 2^{AP}$ be a finite *alphabet*. A letter a in Σ is interpreted as assigning truth values to the elements of AP; i.e., elements in a are

[1] In Section 4.2, we compute the longest sampling period of a CFG based on BCET of basic blocks. This computation is quite realistic, as (1) all hardware vendors publish the BCET of their instruction set in terms of clock cycles, and (2) BCET is a conservative approximation and no execution occurs faster than that.

```
1:      a = scanf(...);
2:      if (a % 2 == 0) goto 9
3:      else {
4:            printf(a + "is odd");
5:*           b = a/2;
6:*           c = a/2 + 1;
7:            goto 10;
8:      }
9:      printf(a + "is even");
10:     end program
```

(a) A simple C program



(b) Control-flow graph

Figure 4.1: A C program and its control-flow graph

assigned true (denoted $\top$) and elements not in a are assigned false (denoted $\bot$). A *timed word* over Σ is a sequence $(a_0, t_0), (a_1, t_1) \cdots (a_k, t_k)$, where each $a_i \in \Sigma$ and each t_i is in non-negative real numbers $\mathbb{R}_{\geq 0}$ and the occurrence times increase monotonically. Let X be a set of *clock variables*. A *clock constraint* over X is a Boolean combination of formulae of the form $x \preceq c$ or $x - y \preceq c$, where $x, y \in X$, $c \in \mathbb{Z}_{\geq 0}$, and $\preceq$ is either $<$ or $\leq$. We denote the set of all clock constraints over X by $\Phi(X)$. A *clock valuation* is a function $\nu : X \to \mathbb{R}_{\geq 0}$ that assigns a real value to each clock variable. For $\tau \in \mathbb{R}_{\geq 0}$, we write $\nu + \tau$ to denote $\nu(x) + \tau$ for every clock variable x in X. Also, for $\lambda \subseteq X$, $\nu[\lambda := 0]$ denotes the clock valuation that assigns 0 to each $x \in \lambda$ and agrees with ν over the rest of the clock variables in X.

Definition 2 (Timed Automaton) *A* timed automaton *is a tuple* $\mathcal{A} = \langle L, L^0, X, \Sigma, E, I \rangle$, *where*

- L *is a finite set of* locations.
- $L^0 \subseteq L$ *is a set of* initial locations.
- X *is a finite set of clock variables.*
- Σ *is a finite set of labels.*
- $E \subseteq (L \times \Sigma \times 2^X \times \Phi(X) \times L)$ *is a set of* switches. *A switch* $\langle l, a, \lambda, \varphi, l' \rangle$ *represents a transition from location* l *to location* l' *labelled by* a, *under clock constraint* φ. *The set* $\lambda \subseteq X$ *gives the clocks to be reset with this switch.*

- $I : L \to \Phi(X)$ *assigns a* delay invariant *to a location.* □

The semantics of a timed automaton $\mathcal{A}$ is as follows. A *state* is a pair (l, ν), where $l \in L$ and ν is a clock valuation for X. A state (l, ν) is an initial state if $l \in L^0$ and $\nu(x) = 0$ for all $x \in X$. There are two types of *transitions*:

1. *Location switches* are of the form $\langle l, a, \lambda, \varphi, l' \rangle$, such that ν satisfies φ, $(l, \nu) \xrightarrow{a} (l', \nu[\lambda := 0])$, and $\nu[\lambda := 0]$ satisfies $I(l')$.
2. *Delay transitions* are of the form $(l, \nu) \xrightarrow{\tau} (l, \nu + \tau)$, which preserves the location l for time duration $\tau \in \mathbb{R}_{\geq 0}$, such that for all $0 \leq \tau' \leq \tau$, $\nu + \tau'$ satisfies the delay invariant $I(l)$.

For a timed word $w = (a_0, t_0), (a_1, t_1) \cdots (a_k, t_k)$, a *run* over w is a sequence

$$q_0 \xrightarrow{t_0} q'_0 \xrightarrow{a_0} q_1 \xrightarrow{t_1 - t_0} q'_1 \xrightarrow{a_1} q_2 \xrightarrow{t_2 - t_1} q'_2 \xrightarrow{a_2} q_3 \to \cdots \xrightarrow{a_k} q_{k+1}$$

such that q_0 is an initial state.

Let $\mathcal{A}_1 = \langle L_1, L_1^0, X_1, \Sigma_1, E_1, I_1 \rangle$ and $\mathcal{A}_2 = \langle L_2, L_2^0, X_2, \Sigma_2, E_2, I_2 \rangle$ be two timed automata, where $X_1 \cap X_2 = \emptyset$. The *parallel composition* of $\mathcal{A}_1$ and $\mathcal{A}_2$ is $\mathcal{A}_1 || \mathcal{A}_2 = \langle L_1 \times L_2, L_1^0 \times L_2^0, X_1 \cup X_2, \Sigma_1 \cup \Sigma_2, E, I \rangle$, where $I(l_1, l_2) = I(l_1) \ \wedge \ I(l_2)$, and E is defined by:

1. for $a \in \Sigma_1 \cap \Sigma_2$, for every $\langle l_1, a, \lambda_1, \varphi_1, l'_1 \rangle$ in E_1, and $\langle l_2, a, \lambda_2, \varphi_2, l'_2 \rangle$ in E_2, E contains $\langle (l_1, l_2), a, \lambda_1 \cup \lambda_2, \varphi_1 \wedge \varphi_2, (l'_1, l'_2) \rangle$.
2. for $a \in \Sigma_1 \backslash \Sigma_2$, for every $\langle l, a, \lambda, \varphi, l' \rangle$ in E_1, and every $m \in L_2$, E contains $\langle (l, m), a, \lambda, \varphi, (l', m) \rangle$.
3. for $a \in \Sigma_2 \backslash \Sigma_1$, for every $\langle l, a, \lambda, \varphi, l' \rangle$ in E_2, and every $m \in L_1$, E contains $\langle (m, l), a, \lambda, \varphi, (m, l') \rangle$.

Figure 4.4(a) shows the timed automaton of the control-flow graph presented in Figure 4.1(b).

4.1.3 3-Valued Linear Temporal Logic (LTL)

Linear temporal logic (LTL) [90] is a popular formalism for specifying properties of (concurrent) programs. The set of well-formed linear temporal logic formulas is constructed from a set of atomic propositions, the standard Boolean operators, and temporal operators. A *word* is a finite or infinite sequence of letters $w = a_0 a_1 a_2 \ldots$, where $a_i \in \Sigma$ for all $i \geq 0$. We denote the set of all finite words over Σ by Σ^* and the set of all infinite words by Σ^ω. For a finite word u and a word w, we write $u \cdot w$ to denote their *concatenation*.

Definition 3 (LTL Syntax) LTL *formulas are defined inductively as follows:*

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathbf{U} \varphi_2$$

where $p \in \Sigma$, and $\bigcirc$ (next) and $\mathbf{U}$ (until) are temporal operators. □

Definition 4 (LTL Semantics) *Let $w = a_0 a_1 \ldots$ be an infinite word in Σ^ω, i be a non-negative integer, and $\models$ denote the* satisfaction *relation. Semantics of* LTL *is defined inductively as follows:*

$$\begin{array}{lll}
w, i \models \top & & \\
w, i \models p & \textit{iff} & p \in a_i \\
w, i \models \neg\varphi & \textit{iff} & w, i \not\models \varphi \\
w, i \models \varphi_1 \vee \varphi_2 & \textit{iff} & w, i \models \varphi_1 \ \vee \ w, i \models \varphi_2 \\
w, i \models \bigcirc\varphi & \textit{iff} & w, i+1 \models \varphi \\
w, i \models \varphi_1 \,\mathbf{U}\, \varphi_2 & \textit{iff} & \exists k \geq i : w_k \models \varphi_2 \ \wedge \ \forall i \leq j \leq k : w, j \models \varphi_1
\end{array}$$

In addition, $w \models \varphi$ holds iff $w, 0 \models \varphi$ holds. □

In order to reason about correctness of programs with respect to an LTL property, we describe a program in terms of its state space and transitions. A *program* is a tuple $P = \langle S, T \rangle$, where S is the non-empty *state space* and T is a set of *transitions*. A transition is of the form (s_0, s_1), where $s_0, s_1 \in S$. A state of a program is normally obtained by evaluation of its variables and transitions are program instructions. In this context, an atomic proposition in a program is a Boolean predicate over S (i.e., a subset of S).

We define a *trace* of a program $P = \langle S, T \rangle$ by a finite or infinite sequence of states $\sigma = s_0 s_1 s_2 \ldots$, such that $s_i \in S$ and each $(s_i, s_{i+1}) \in T$, for all $i \geq 0$. A program trace σ *satisfies* an LTL property φ (denoted $\sigma \models \varphi$) iff $\sigma \in L(\varphi)$. If σ does not satisfy φ, we say that σ *violates* φ. A program P satisfies an LTL property φ (denoted $P \models \varphi$) iff for each trace σ of P, $\sigma \models \varphi$ holds. For example, consider the following piece of code:

```
x := 10;
while (x <= 100) {
x := x + 2;
}
```

It is straightforward to observe that this code satisfies the following properties: $\Box p$ and $\Diamond q$, where $p \equiv (x \geq 10)$ and $q \equiv (x = 100)$. Note that all temporal operators in LTL (i.e., $\Box$, $\Diamond$, $\mathbf{R}$) can be presented via operators $\mathbf{U}$ and $\bigcirc$. For instance, $\Diamond p$ (i.e., eventually p) $= true\mathbf{U}p$, $\Box p$ (i.e., always p) $= \neg(true\mathbf{U}\neg p)$, and $p\mathbf{R}q$ (i.e., q remains true until once p becomes true. p may never become true) $= \neg(\neg p\mathbf{U}\neg q)$.

Implementing runtime verification boils down to the following problem: given a *finite* word $\sigma = a_0 a_1 a_2 \dots a_n$, check whether or not σ belongs to a set of words defined by some property φ. This is a complex problem, because LTL semantics is defined over infinite words and a running program can only deliver a finite word at a verification point (i.e., monitor sample). For example, given a finite word $\sigma = a_0 a_1 \dots a_n$, it may be impossible for a *monitor* to decide weather the property $\Diamond p$ is satisfied. To clarify this issue, consider the case where for all i, $0 \leq i \leq n$, $p \notin s_i$, then the program still has the chance to satisfy $\Diamond p$ in the future. In other words, a monitor can decide whether a property is violated only if all the reachable states of the program under inspection cannot possibly satisfy the property. Respectively, the monitor reports satisfaction of the property if any sequence of program states that follows σ satisfies the property.

To formalize satisfaction of LTL properties at run time, in [9], the authors propose semantics for LTL, where the evaluation of a formula ranges over three values '$\top$', '$\bot$', and '?' (denoted LTL_3). The latter value expresses the fact that it is not possible to decide on the satisfaction of a property, given the current finite trace of the program.

Definition 5 (LTL_3 semantics) *Let $u \in \Sigma^*$ be a finite word. The truth value of an LTL_3 formula φ with respect to u, denoted by $[u \models \varphi]$, is defined as follows:*

$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall w \in \Sigma^\omega : u \cdot w \models \varphi, \\ \bot & \text{if } \forall w \in \Sigma^\omega : u \cdot w \not\models \varphi, \\ ? & \text{otherwise.} \end{cases}$$

□

Note that the syntax $[u \models \varphi]$ for LTL_3 semantics is defined over finite words as opposed to $u \models \varphi$ for LTL semantics, which is defined over infinite words. For example, given a

finite program trace $\sigma = a_0 a_1 \cdots a_n$, property $\Diamond p$ holds iff $a_i \models p$, for some i, $0 \leq i \leq n$ (i.e., σ is a good prefix). Otherwise, the property evaluates to ?. Thus, one can reason about satisfaction of properties in LTL_3 through the notion of *good/bad* prefixes defined next.

Definition 6 (Good and Bad Prefixes) *Given a language $L \subseteq \Sigma^\omega$ of infinite words over Σ, we call a finite word $u \in \Sigma^*$*

- *a* good prefix *for L, if $\forall w \in \Sigma^\omega : u \cdot w \in L$*
- *a* bad prefix *for L, if $\forall w \in \Sigma^\omega : u \cdot w \notin L$*
- *an* ugly *prefix otherwise.* □

Implementing runtime verification for an LTL_3 property involves synthesizing a monitor that realizes the property. In [9], the authors introduce a stepwise method that takes an LTL_3 property φ as input and generates a deterministic finite state machine (FSM) $\mathcal{M}^\varphi$ as output. Intuitively, simulating a finite word u on this FSM reaches a state that illustrates the valuation of $[u \models \varphi]$.

Definition 7 (Monitor) *Let φ be an LTL_3 formula over alphabet Σ. The* monitor *$\mathcal{M}^\varphi$ of φ is the unique FSM $(\Sigma, Q, q_0, \delta, \lambda)$, where Q is a set of states, q_0 is the initial state, δ is the transition relation, and λ is a function that maps each state in Q to a value in $\{\top, \bot, ?\}$, such that:*

$$[u \models \varphi] = \lambda(\delta(q_0, u)).$$

□

For example, consider the property $\varphi \equiv (\neg \mathit{spawn}\ \mathbf{U}\ \mathit{init})$ (i.e., a thread is not spawned until it is initialized). The corresponding monitor is shown in Figure 4.2 [9]. The proposition *true* denotes the set AP of all propositions. We use the term a *conclusive state* to refer to monitor states $q_\top$ and $q_\bot$; i.e., a state where $\lambda(q) = \top$ and $\lambda(q) = \bot$, respectively. Other states are called *inconclusive states*. A monitor $\mathcal{M}^\varphi$ is constructed in a way that it recognizes good, bad, and ugly prefixes of $L(\varphi)$. Hence, a conclusive state is in fact also a *trap* state. In other words, if $\mathcal{M}^\varphi$ reaches a conclusive state, it stays in this state.

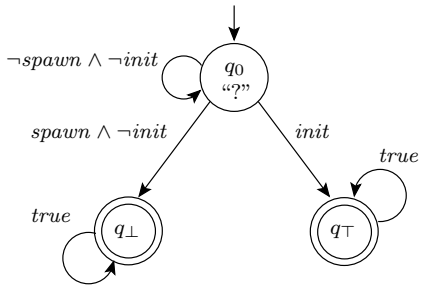


Figure 4.2: The monitor for property $\varphi \equiv (\neg spawn\ \mathbf{U}\ init)$

4.2 Semantics of Time-Triggered Monitor

Given a program P, we describe the semantics of time-triggered monitoring in two steps.

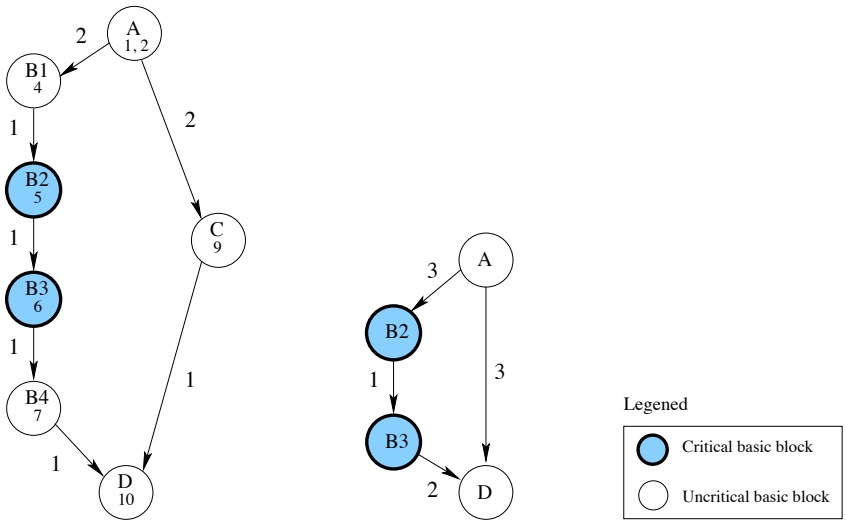
1. Identifying the longest sampling period.
2. Constructing a time-triggered monitor and composing it with P.

Then, we show that the obtained composition can effectively verify a rich fragment of LTL_3 properties at run time.

4.2.1 Calculating the Longest Sampling Period

Let P be a program and Π be an LTL_3 property, where P is expected to satisfy Π. Let $\mathcal{V}_\Pi$ denote the set of variables that can change the valuation of the atomic propositions in Π. For example, in property $\Pi \equiv \Box\Diamond(x \geq 0\ \wedge\ y = 10)$, we have $\mathcal{V}_\Pi = \{x, y\}$. Generally, in our time-triggered monitoring, a *sampler* process periodically wakes up with some sampling period, reads the value of variables in $\mathcal{V}_\Pi$ from program P, and passes them to a monitor $\mathcal{M}^\varphi$ (as described in Subsection 4.1.3) to evaluate Π. As can be seen, the sampler process is the observer component that is invoked periodically[2]. The sampler process is discussed in more detail in Subsection 4.2.2. The main challenge in this mechanism is accurate reconstruction of the states that P takes in between two consecutive samples from the sampler process. For instance, if the value of a variable in $\mathcal{V}_\Pi$ changes more than once

[2]Note that in the case of the sampler process, the observer invokes itself and it is no longer delegated to the program under scrutiny.



(a) Step 1 (b) Step 2

Figure 4.3: Obtaining a critical CFG and calculating the longest sampling period

between two samples, the sampler process can only extract the last value of the variable. Hence, the monitor may fail to evaluate property Π correctly.

In order to accurately sample all the changes in the value of variables in $\mathcal{V}_\Pi$, we modify CFG_P as follows. In the first step, we ensure that each *critical instruction* (i.e., an instruction that modifies the value of a variable in $\mathcal{V}_\Pi$) is in a basic block that contains no other instructions. We refer to such a basic block as *critical basic block* or *critical vertex*. Formally, let $inst_v = \langle v^1 \cdots v^n \rangle$ denote the sequence of instructions in a basic block v of CFG_P. Let v^i, where $1 < i < n$, be the one and only critical instruction in $inst_v$. We split vertex v into three vertices v_1, v_2, and v_3, such that $inst_{v_1} = \langle v_1^1 \cdots v_1^{i-1} \rangle$, $inst_{v_2} = \langle v_2^i \rangle$, and $inst_{v_3} = \langle v_3^{i+1} \cdots v_3^n \rangle$. Incoming arcs to v now enter v_1. We add arc (v_1, v_2), where $w(v_1, v_2)$ is equal to the best-case execution time of $\langle v_1^1 \cdots v_1^{i-1} \rangle$. We also add arc (v_2, v_3), where $w(v_2, v_3)$ is equal to the best-case execution time of $\langle v_2^i \rangle$. Outgoing arcs from v now leave v_3 with weight equal to the best-case execution time of $\langle v_3^{i+1} \cdots v_3^n \rangle$. Obviously, if $i = 1$ or $i = n$, we split v into two vertices. We continue this procedure until each critical instruction is in a separate basic block. For example, in the program in Figure 4.1(a), if variables `b` and `c` are in $\mathcal{V}_\Pi$, then instructions 5 and 6 are critical, and hence, we obtain the control-flow graph in Figure 4.3(a).

Since non-critical vertices do not play a role in determining the sampling period, in the second step, our method collapses non-critical vertices as follows. Let $CFG = \langle V, v^0, A, w \rangle$

be a control-flow graph. *Transformation* $T(CFG, v)$, where $v \in V \backslash \{v^0\}$ and outdegree of v is positive, obtains $CFG' = \langle V', v^0, A', w' \rangle$ via the following ordered steps:

1. Let A'' be the set $A \cup \{(u_1, u_2) \mid (u_1, v), (v, u_2) \in A\}$. Observe that if an arc (u_1, u_2) already exists in A, then A'' will contain parallel arcs (such arcs can be distinguished by a simple indexing or renaming scheme). We eliminate the additional arcs in Step 3.

2. For each arc $(u_1, u_2) \in A''$,
$$w'(u_1, u_2) = \begin{cases} w(u_1, u_2) & \text{if } (u_1, u_2) \in A \\ w(u_1, v) + w(v, u_2) & \text{if } (u_1, u_2) \in A'' \backslash A \end{cases}$$

3. If there exist parallel arcs from vertex u_1 to u_2, we only include the one with minimum weight in A''.

4. Finally, $A' = A'' \backslash \{(u_1, v), (v, u_2) \mid u_1, u_2 \in V\}$ and $V' = V \setminus \{v\}$.

We clarify a special case of the above transformation, where u and v are two non-critical vertices with arcs (u, v) and (v, u) between them. Deleting one of the vertices, e.g., u, results in a self-loop (v, v), which we can safely remove. This is simply because a loop that contains no critical instructions has no effect on the calculation of the longest sampling period.

We apply the above transformation on all non-critical vertices. We call the resulting graph a *critical control-flow graph*. Such a graph includes (1) a non-critical initial basic block, (2) possibly a non-critical vertex with outdegree zero (if the program is terminating), and (3) a set of critical vertices. Figure 4.3(b) shows the critical control-flow graph in Figure 4.3(a).

Definition 8 (Longest Sampling Period) *Let $CFG = \langle V, v^0, A, w \rangle$ be a critical control-flow graph. The* longest sampling period *(LSP) for CFG is*
$$LSP_{CFG} = \min\{w(v_1, v_2) \mid (v_1, v_2) \in A \ \wedge \ v_1 \text{ is a critical vertex}\}$$

□

Intuitively, the longest sampling period is the minimum time interval between the execution of two critical instructions that change the value of a variable in $\mathcal{V}_\Pi$. For example, the longest sampling period of the control-flow graph in Figure 4.3(b) is $LSP = 1$. Later in this section, we show that by applying this sampling period, one can verify the correctness of a rich fragment of LTL_3 properties at run time.

4.2.2 Constructing and Composing a Time-triggered Monitor

We now explain the semantics of time-triggered monitoring using timed automata. We note that our implementation does not explicitly use the transformation presented in this subsection; i.e., we solely use the timed automata formalism to describe the semantics and our implementation is a refinement of the transformation. Transformation of a control-flow graph $CFG = \langle V, v^0, A, w\rangle$ into a timed automaton $\mathcal{A}_{CFG} = \langle L, L^0, X, \Sigma, E, I\rangle$, where $X = \{t\}$ and $\Sigma = \{a, s\}$, is as follows:

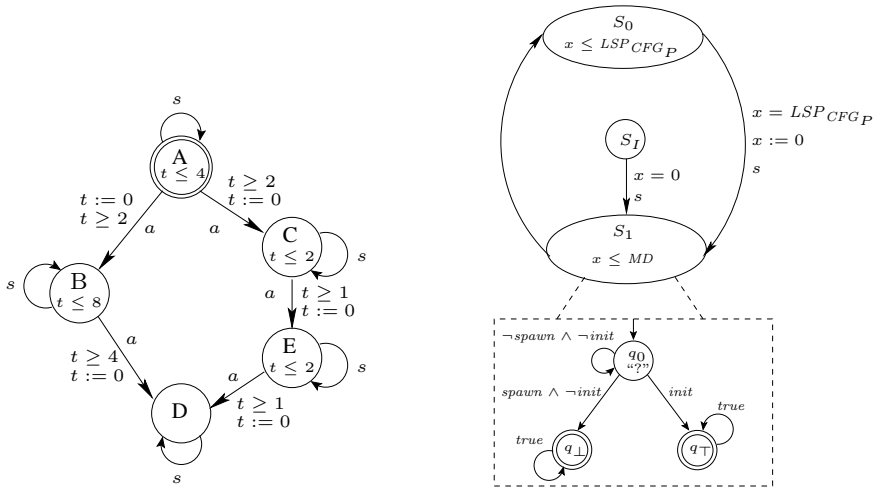
- $L = \{l_v \mid v \in V\}$
- $L^0 = \{l_{v^0}\}$
- $E = \{\langle l_v, a, \{t\}, t \geq w(v, v'), l_{v'}\rangle \mid (v, v') \in A\} \cup \{\langle l_v, s, \{\}, \mathit{true}, l_v\rangle \mid v \in V\}$.
- $I(l_v) =$ worst-case execution time of basic block $v \in V$.

Intuitively, $\mathcal{A}_{CFG}$ works as follows. Each location of $\mathcal{A}_{CFG}$ corresponds to one and only one vertex of CFG. The initial location corresponds to the initial basic block of CFG. Each location is associated with a delay invariant; the execution can stay in a location no longer than the worst-case execution time of the corresponding basic block. $\mathcal{A}_{CFG}$ has two types of switches. The first set of switches (labelled by a) change the location. Each such switch takes place when the execution of the corresponding basic block is complete. Obviously, this can happen not earlier than the best-case execution time of the basic block. The other set of switches (labelled by s) are self-loops and are meant to synchronize with the automaton representing the sampler process. The timed automaton obtained from the control-flow graph in Figure 4.1(b) is shown in Figure 4.4(a), where the worst-case execution time of each instruction is 2.

The relation between execution of a program P and runs of the timed automaton $\mathcal{A}_{CFG_P}$ is as follows. Intuitively, a delay transition in $\mathcal{A}_{CFG_P}$ corresponds to the execution of a set of instructions in P. Formally, let $q = (l, t = 0)$ be a state of $\mathcal{A}_{CFG_P}$, where location l hosts instructions $\{l^1 \cdots l^n\}$. An outgoing transition from this state with delay τ reaches a state $(l, t + \tau)$ which leads to executing zero or more instructions. Thus, starting from $(l, t = 0)$, a run of $\mathcal{A}_{CFG_P}$ is of the form:

$$(l, t = 0) \xrightarrow{\tau_1} (l^i, t + \tau_1) \xrightarrow{\tau_2} (l^j, t + \tau_1 + \tau_2) \xrightarrow{\tau_3} \cdots \xrightarrow{\tau_m} (l^n, t + \textstyle\sum_{k=1}^{m} \tau_k) \xrightarrow{a} (l', t = 0),$$

such that:



(a) Timed automaton of the CFG in Figure 4.1(b)

(b) Time-triggered sampler and monitor

Figure 4.4: Formal semantics of time-triggered monitoring

- $i \leq j \leq m$,
- $l \neq l'$, where (l, l') is a location switch in E,
- $(l^i, t + \tau_1)$ denotes the fact that instructions $\langle l^1 \cdots l^i \rangle$ have been executed within τ_1 time units,
- $\sum_{k=1}^{m} \tau_k \geq w(l, l')$ in CFG_P, and
- $(t \leq \sum_{k=1}^{m} \tau_k) \;\Rightarrow\; I(l)$.

Note that an s-transition may occur in such a run, but such transitions obviously do not change the current location or the value of t. This also holds in practice, because when the sampler process intervenes with the program execution to extract the value of variables in $\mathcal{V}_\Pi$, the execution of the program halts until the sampler process finishes its data extraction and resumes the normal operation of the program.

A sampler process $\mathcal{S}_P$ of a time-triggered monitor for program P works as follows (see Figure 4.4(b)). From the initial location S_I the only outgoing switch is enabled immediately (i.e., when $x = 0$). This switch is labeled by s and the sampler synchronizes with $\mathcal{A}_{CFG_P}$ on the switch in order to read the variables in $\mathcal{V}_\Pi$ and evaluate Π. Consequently, the sampler

reaches location S_1 and may remain in this location for at most MD time units, where MD is the worst-case execution time for reading the variables in $\mathcal{V}_\Pi$ and property evaluation, using the technique presented in Subsection 4.1.3. That is, the sampled program state is simulated on the monitor (e.g., the monitor depicted in location S_1 from Figure 4.2) automaton for property evaluation. We assume that sampling never occurs in the middle of the execution of an instruction. Normally, this assumption is already implemented, as hardware interrupts to generate time ticks are generally handled after completion of fetched instructions. After the verification step in location S_1, the sampler reaches location S_0, where it sleeps until the sampling period is complete (i.e, $x = LSP_{CFG_P}$). Thus, the parallel composition $\mathcal{A}_{CFG_P}||\mathcal{S}_P$ constructs the entire monitored system. For example, the following is a run of the automaton in Figure 4.4(a), composed with a sampler with sampling period $LSP = 1$ and $MD = 0$:

$$AS_I \xrightarrow{s} AS_1 \to AS_0 \xrightarrow{1} A^1S_0 \xrightarrow{s}$$

$$A^1S_1 \to A^1S_0 \xrightarrow{1} A^2S_0 \xrightarrow{s} A^2S_1 \to A^2S_0 \xrightarrow{a} BS_0 \xrightarrow{1} B^4S_0 \to \cdots,$$

We call the combination of such a time-triggered sampler and a monitor (as illustrated in Figure 4.4(b)) a *time-triggered monitor*.

4.2.3 Correctness of Time-triggered Runtime Monitoring

In this section, we show that runtime verification using a time-triggered monitor is sound for a rich fragment of LTL$_3$.

Assumption 1 *We assume that $MD \leq LSP$.* □

This assumption is quite realistic. That is, the time a time-triggered monitor needs to read the state of the program and evaluate properties is less than the sampling period. This can be, for instance, guaranteed by scheduling all verification tasks at run time on a different computing core. We now show that our monitor construction method is sound with respect to observing all value changes of variables.

Lemma 1 *Let P be a program and $w = (a_0, t_0), (a_1, t_1) \cdots$ be a timed word of $\mathcal{A}_{CFG_P}||\mathcal{S}_P$. For all i and j, where $i < j$, $a_i = a_j = s$, and there does not exist an s-transition between a_i and a_j in w, no run over w contains delay transitions between a_i and a_j that includes two critical instructions.*

Proof The lemma holds by construction of $\mathcal{S}_P$, as it enforces sampling period LSP. We only describe three cases for the sake of clarity:

- Note that if all locations of $\mathcal{A}_{CFG_P}$ show their worst-case execution time, the monitor still observes all critical state changes. One can think of this scenario similar to a sliding window with fixed size (equal to LSP) that can move over a run. Since the window can never observe two critical state changes, worst-case executions are irrelevant to sampling points.
- The above argument also clarifies why the delay invariant of location S_1 in $\mathcal{S}_P$ causes no incorrectness.
- Finally, removing self-loops from non-critical vertices do not create any problems, since those loops do not contain critical instructions. Thus, no matter how many times such loops iterate, the longest sampling period guarantees correctness. □

Lemma 1 has several consequences important to deploying and running a time-triggered monitor:

- Once a time-triggered monitor starts its execution, it can soundly re-construct the state of the inspected program, regardless of the time the time-triggered monitor started executing. This implies that even when the time-triggered monitor has execution offset from the execution start of the inspected program, the time-triggered monitor starts sampling the program correctly as soon as the it starts its execution. Thus, if a time-triggered monitor crashes and recovers, it will work correctly from the point it restarts sampling.
- If the inspected program does not exhibit best-case execution time (which is normally the case), then the program executes at a slower pace. In this case, the time-triggered monitor still samples the program with LSP and this ensures that no critical instructions are overlooked. Thus, if the inspected program is later augmented by new code that does not decrease the minimum time interval between the execution of two consecutive critical instructions and hence, there is no need to change the sampling period or the time-triggered monitor structure.
- Unlike worst/average-case execution, best-case execution time analysis is a straightforward procedure. Most hardware vendors publish the best-case execution time of instructions sets based on CPU clock cycles. Thus, our method for constructing a time-triggered monitor is a conservative, but robust approach for deployment.

A valid question in the context of our approach is whether any LTL$_3$ property can be soundly verified when the time-triggered sampler is in location S_1. To intuitively answer this question, first, consider the following property $\Pi_1 \equiv \Diamond p$; i.e., eventually proposition p holds. Since the sampler reads the state of the program after any change in the value of variables in proposition p (i.e., variables in $\mathcal{V}_\Pi$), when it reaches state S_1, simulation of proposition p on $\mathcal{M}^{\Pi_1}$ can trivially determine whether the valuation of φ is '?' or conclusive.

On the other hand, consider LTL property $\Pi_2 \equiv (p \Rightarrow \bigcirc q)$; i.e., if p holds, then proposition q holds in the next state. In this case, a time-triggered monitor cannot soundly evaluate Π_2. The reason is simple: the sampler only reads the state of the program when it wakes up. Thus, if proposition p becomes true in the next program state, the sampler may wake up several states after q becomes true. Hence, when the sampler wakes up, if q is false, then the monitor can evaluate Π_2 to $\bot$. However, if q is true, then the monitor cannot deduce whether q became true in the immediate state after p becomes true, or in some other state.

In the next theorem, we show that a time-triggered monitor is sound for verification of the fragment of LTL$_3$ which excludes the next operator (denoted LTL$_3^{-\bigcirc}$). To this end, we first set our terminology based on standard concepts. A *state* of a program is a valuation of its variables. Notice that each state of a program determines a set of propositions that hold in that state. Thus, a finite word of a program is trivially defined by a finite sequence of states of the program. Given a finite word u of a program, where each letter in u is read by the time-triggered monitor, we denote the complete finite word of the program by $\hat{u}$. Formally, let $u = a_0 a_1 a_2 \cdots a_n$ and $\hat{u} = b_0 b_1 b_2 \cdots b_m$. If a_i and a_{i+1} are two letters in u, where $0 \leq i \leq n$, then (1) there exist j and k, such that $0 \leq j \leq k \leq m$, $b_j = a_i$, and $b_k = a_{i+1}$, and (2) if there exists l, where $j < l < k$, then state b_l is not sampled by the time-triggered monitor.

Lemma 2 *Let p be a proposition in AP and $u = a_0 a_1 \cdots a_n$ be a finite word of a program P that is sampled by a time-triggered monitor with sampling period LSP. Let $\hat{u} = b_0 b_1 \cdots b_m$. It is the case that $p \notin b_i$ and $p \in b_{i+1}$, where $0 \leq i \leq m$, if and only if there exists j, $0 \leq j \leq n$, such that $p \notin a_j$ and $p \in a_{j+1}$.*

Proof The proof follows trivially from Lemma 1. □

Intuitively, Lemma 2 shows that if a proposition becomes true in a state, then the time-triggered monitor always detects it in the next immediate sample.

Lemma 3 *Let p and q be two propositions in AP and $u = a_0 a_1 \cdots a_n$ be a finite word of a program P that is sampled by a time-triggered monitor with sampling period LSP. Let $\hat{u} = b_0 b_1 \cdots b_m$. It is the case that $p \in b_i$ and $q \in b_j$, where $0 \leq i \leq j \leq m$ if and only if there exists i' and j', $0 \leq i' \leq j' \leq n$, such that $p \in a_{i'}$ and $p \in a_{j'}$.*

Proof The proof follows trivially from Lemma 2. □

Intuitively, Lemma 3 shows that the causal order of occurrence of events when detected by a time-triggered monitor is correct.

Theorem 1 *Let P be a program, Π be a property in $\text{LTL}_3^{-\bigcirc}$, and u be a finite word of P that is sampled by a time-triggered monitor with sampling period LSP and $\hat{u}$ be the corresponding complete finite word. It is the case that $[u \models \Pi] = [\hat{u} \models \Pi]$.*

Proof Let $u = a_0 a_1 \cdots a_n$ and $\hat{u} = b_0 b_1 \cdots b_m$. We prove the theorem in an inductive fashion:

- Let $\Pi \equiv p$, where p is an atomic proposition and $[\hat{u} \models p] = \top$. This implies that $p \in b_0$. Since the time-triggered monitor samples the program in the initial state (i.e., the switch from location S_I to S_1 in Figure 4.4(b)), we have $a_0 = b_0$. Thus, the monitor can determine the truthfulness of $p \in a_0$. The same argument holds for other possible values of $[u \models p]$. Also, the same claim can be proved for any point of a finite word other than a_0 and b_0, in the same fashion by applying Lemma 2.
- For cases, where the property is of the form $\neg\Pi$ or $\Pi_1 \wedge \Pi_2$, the proof is implied by Lemma 2 and is identical to the proof of the previous case.
- Let $\Pi \equiv \Pi_1 \mathbf{U} \Pi_2$. We now show that $[u \models \Pi] = [\hat{u} \models \Pi]$. We distinguish three sub-cases:
  - Let us assume that $[\hat{u} \models \Pi_1 \mathbf{U} \Pi_2] =?$. It follows that (1) there does not exist $k \leq m$, such that $[\hat{u}, k \models \Pi_2] = \top$, and (2) for all i, $0 \leq i \leq m$, $[\hat{u}, i \models \Pi_1] = \top$. By applying Lemmas 2 and 3, it is straightforward to see that there does not exist $l \leq n$, such that $[u, l \models \Pi_2] = \top$, and (2) for all j, $0 \leq j \leq n$, $[u, j \models \Pi_1] = \top$. Hence, we have $[u \models \Pi_1 \mathbf{U} \Pi_2] =?$.

– Let us assume that $[\hat{u} \models \Pi_1 \mathbf{U} \Pi_2] = \top$. It follows that (1) there exists $k \leq m$, such that $[\hat{u}, k \models \Pi_2] = \top$, and (2) for all i, $0 \leq i \leq k$, $[\hat{u}, i \models \Pi_1] = \top$. By applying Lemmas 2 and 3, it is straightforward to see that there exists $l \leq n$, such that $[u, l \models \Pi_2] = \top$, and (2) for all j, $0 \leq j \leq n$, $[u, j \models \Pi_1] = \top$. Hence, we have $[u \models \Pi_1 \mathbf{U} \Pi_2] = \top$.

– Let us assume that $[\hat{u} \models \Pi_1 \mathbf{U} \Pi_2] = \bot$. It follows that (1) there does not exist $k \leq m$, such that $[\hat{u}, k \models \Pi_2] = \top$, or (2) there exists i, $0 \leq i \leq m$, $[\hat{u}, i \models \Pi_1] = \bot$. By applying Lemmas 2 and 3, it is straightforward to see that there does not exist $l \leq n$, such that $[u, l \models \Pi_2] = \top$, or (2) there exists j, $0 \leq j \leq n$, $[u, j \models \Pi_1] = \bot$. Hence, we have $[u \models \Pi_1 \mathbf{U} \Pi_2] = \bot$. □

Chapter 5

Optimizing the Longest Sampling Period

Initial studies on programs from the SNU [2] and MiBench [1] bench suites show that typically the longest sampling period (see Subsection 4.2.1) has a small value with respect to the execution time of the program. To this end, it results in highly frequent monitoring samples in the program execution at run time. As a result, we hypothesized that the time-triggered monitor imposes large overall overhead at run time.

Recall from Chapter 3 that in this thesis we only focus on the sampler (i.e., the observer module) of a runtime verification framework. We assume that the monitor itself (i.e., verification engine) runs asynchronously on a separate machine. Hence, the overhead imposed by the verification process does not affect the execution of the program under scrutiny. To this end, in the remaining of this thesis, *monitoring overhead* only refers to the overhead imposed by the sampler at run time.

With this respect, experiments from Section 5.3.3 still support our hypothesis which show that on average the time-triggered monitor can impose over 170% monitoring overhead at run time. Thus, our time-triggered monitor does not provide *efficient monitoring* which is required for runtime verification of real-time embedded systems. To achieve efficient monitoring, one must reduce the monitoring samples. Hence, we aim at solving the following problem.

Problem statement. Given a set of properties Π (e.g. LTL formulas) and a program P, increase the longest sampling period (LSP) such that the monitor provides sound runtime verification (i.e., samples all of the critical instructions).

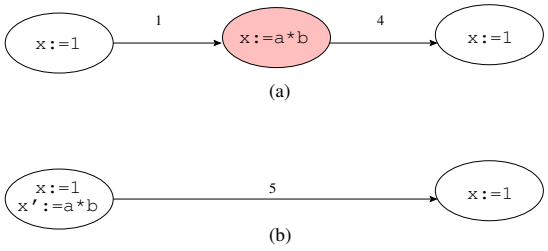


Figure 5.1: IT transformation applied on the middle basic block

In this chapter, we present the *history approach* [21] [19] which uses auxiliary memory to solve the aforementioned program (Problem 5). The history approach uses auxiliary memory to build a history of program state changes between two monitoring samples. Section 5.1 presents the history approach [21] [19], Section 5.2 presents the complexity of finding the optimal solution to Problem 5 by using the history approach [19], Section5.3 uses integer linear programming to optimally solve Problem 5 [21] [19], and Section 5.4 presents three near-optimal solutions to Problem 5 [85].

5.1 The History Approach

The history based approach uses auxiliary memory to store the outcome of critical instructions and henceforth, safely increase the longest sampling period. More specifically, let (u, v) be an arc and v be a vertex in a critical control-flow graph CFG, where $inst_v = \langle i \rangle$ and i changes the value of a variable, say a. We apply transformation $T(CFG, v)$ introduced in Subsection 4.2.1 and add an instruction $i' : a' \leftarrow a$, where a' is an auxiliary memory location. Thus, we obtain $inst_u = inst_u.\langle i, i' \rangle$. We call this process *instrumenting transformation* and denote it by $IT(CFG, v)$. For example, in Figure 5.1(a), assuming that `x` is a variable in $\mathcal{V}_\Pi$, all three basic blocks are critical. Thus, the longest sampling period is 1 time unit. Figure 5.1(b) shows the control-flow graph obtained by applying the IT transformation on the shaded vertex, where the corresponding instruction is added to the previous vertex and the value of `x` is stored in `x'`.

Unlike non-critical vertices, the issue of loops involving critical vertices need to be handled differently. Suppose u and v are two critical vertices with arcs (u, v) and (v, u) between them and we intend to apply IT on vertex u. This results in a self-loop (v, v), where $w(v, v) = w(u, v) + w(v, u)$. Since we do not know how many times the loop may iterate at run time, it is impossible to determine the upperbound on the size of auxiliary

memory needed to collapse vertex v. Hence, to ensure correctness, we do not allow applying transformation IT on critical vertices that have self-loops.

Before we elaborate on the formulation of optimal instrumentation of a program, two issues need to be addressed with regard to instrumenting a program using the IT transformation. The first issue is whether instrumenting a program affects the calculation of the longest sampling period as described in Subsection 4.2.1. Observe that adding an extra instruction to a critical basic block only extends the execution time of that basic block. As argued in Section 4.2, if the execution of a basic block gets extended for any reason, it does not affect the correctness of a time-triggered monitor. This is due to the fact that adding instrumentation only increases the best-case execution time of a basic block and by maintaining the calculated sampling period, we are guaranteed that no critical instruction is overlooked.

The second issue is whether the transformed program inspected by a time-triggered monitor exhibits the same semantics. In particular, let (u, v) be an arc of a critical control-flow graph and v be the basic block on which transformation $IT(CFG, v)$ is applied. Let the critical instruction in v be $inst_v = \langle i \rangle$, which updates a variable a and the added instruction by IT be $i' : a' \leftarrow a$. Let us assume that variable a participates in valuation of a proposition p. In this setting, in order to enable a monitor to evaluate LTL_3 properties soundly, one only needs a simple rewriting procedure, so that a property that involves proposition p is rewritten by p' when variable a' is read. Hence, valuation of any $\text{LTL}_3^{-\bigcirc}$property is preserved by applying IT transformations; i.e., the instrumentation instructions do not change the functional properties of the inspected program (obviously with no next operator).

5.2 Complexity Analysis

We now analyze the complexity of achieving optimal instrumentation. Given a critical control-flow graph, our goal is to optimize two factors through a set of IT transformations: (1) minimizing auxiliary memory, and (2) maximizing the longest sampling period. We now analyze the complexity of such optimization.

Instance. A critical control-flow graph $CFG = \langle V, v^0, A, w \rangle$ and positive integers X and Y.

Transformation optimization decision problem (TO). Does there exist a set $U \subseteq V$,

such that after applying transformation $IT(CFG, u)$ for all $u \in U$, we obtain a critical control-flow graph $CFG' = \langle V', v^0, A', w' \rangle$, where $|U| \leq Y$ and for all arcs $(u, v) \in A'$, $w'(u, v) \geq X$?

Theorem 2 *TO is NP-complete.*

Proof Since showing membership to NP is straightforward, we only need to prove that TO is NP-hard. To this end, we reduce the *Minimum Vertex Cover Problem* (VC) [56] to TO. The minimum vertex cover problem is as follows: Given a (directed or undirected) graph $G = \langle V, E \rangle$ and a positive integer K, the problem is to find a set $V' \subseteq V$, such that $|V'| \leq K$ and each edge in E is incident to at least one vertex in V'.

First, we present a mapping from an instance of VC to an instance of TO. Then, we illustrate a reduction using our mapping.

Mapping. Let digraph $G = \langle V_1, E \rangle$ and positive integer K be an arbitrary instance of VC. We obtain an instance of TO as follows:

- We construct digraph $CFG = \langle V_2, v^0, A, w \rangle$ as follows:
  - $V_2 = V_1 \cup \{v^0\}$, where v^0 is an additional vertex representing the initial basic block of CFG.
  - $A = E \cup \{(v^0, u) \mid u \in V_2\}$,
  - $w(v_0, u) = 2$ for all $u \in V_2$ and $w(v, u) = 1$, for all $v \in V_2 - \{v^0\}$.
- Finally, we let $Y = K$ and $X = 2$.

Reduction. Now, we show that the answer to an instance of VC is affirmative if and only if the answer to TO is positive:

- ($\Rightarrow$) Let $V_1' \subseteq V_1$ be the answer to VC for G, such that $|V_1'| \leq K$. We now show that the set V_2' identical to V_1' is the answer to TO. First, observe that $|V_2'| \leq Y$. Now, notice that deleting a vertex in V_2' results in all pairs of incoming and outgoing arcs to be replaced by edges of weight 2. The only case where an edge of weight 2 is not created between two vertices, say u and v, is when an edge of cost 1 already exists between u and v. However, since all arcs are covered by a vertex in V_2', the arc with

weight 1 will be replaced by an arc of weight at least 2 through another vertex in V_2' as well. Finally, since all vertices have indegree and outdegree of at least 1, all arcs are replaced by arcs of cost at least 2.

- ($\Leftarrow$) Let $V_2' \subseteq V_2$ be the answer to TO, such that $|V_2'| \leq Y$. We now show that the set V_1' identical to V_2' is the answer to VC. First, observe that $|V_1'| \leq K$. Now, since the weight of all arcs in A are at least 2, all edges in E_1 must be incident to at least one vertex in V_1'. This simply implies that V_1' is a cover for E_1. □

Obviously, time-triggered monitoring and in particular, increasing the sampling period introduces *detection latencies*. Detection latency represents the maximum amount of time between the point at which the program violates a property by executing a corrupting critical instruction to the point where the monitor is invoked and carries out verification. To tackle this problem, one can specify a *tolerable detection delay* for variables in $\mathcal{V}_\Pi$. For instance, the user can specify that for variable $v \in \mathcal{V}_\Pi$, the detection latency should not exceed 5 time units. With this respect, our optimization approach incorporates this restriction when choosing the critical instructions to collapse. In this setting, the approach produces either of the following outputs: (1) achieving the desired sampling period while satisfying the detection latency is not possible, or (2) the set of critical instructions that can be collapsed to achieve the desired sampling period while satisfying the detection latency. This factor can be easily incorporated in our transformation technique and optimization problem.

Theorem 2 clearly shows the tradeoff between minimizing the auxiliary memory size and maximizing the sampling period. For practical reasons, designers may have restrictions over the size of the auxiliary memory for building the histories. Nevertheless, one can increase the sampling period as much as possible to bound the runtime monitoring overhead. The extreme case is to take a sample in the beginning and one at the end of program execution. We now show that building optimized history even for this overly simplified problem remains NP-complete. We denote this problem by MTO.

Theorem 3 *MTO is NP-complete.*

Proof Since showing membership to NP is straightforward, we only need to prove that MTO is NP-hard. To this end, our reduction is from the *Hamiltonian Path Problem* (HP) [56]: Given a (directed or undirected) graph $G = \langle V, E \rangle$, the problem is to determine whether G has a *simple* path that visits all vertices in V.

First, we present a mapping from an instance of HP to an instance of MTO. Then, we illustrate a reduction using our mapping.

Mapping. Let digraph $G_1 = \langle V_1, E_1 \rangle$ be an arbitrary instance of HP. We obtain an instance of MTO as follows:

- We construct digraph $G_2 = \langle V_2, E_2 \rangle$, such that $V_2 = V_1$ and $E_2 = E_1$.
- The cost function is defined as $C(e) = 1$ for all $e \in E_2$
- Finally, we let $Y = |V| - 2$ and $X = |V| - 1$.

Reduction. Now, we show that the answer to an instance of HP is affirmative if and only if the answer to MTO is positive:

- ($\Rightarrow$) If the answer to HP is affirmative, one can delete all vertices except for the first and the last along the Hamiltonian path. It follows that the number of deleted vertices is $|V| - 2$ and since each transformation selects edges with maximum cost, the cost of the final edge is $|V| - 1$.

- ($\Leftarrow$) Suppose that the answer to MTO is affirmative. This implies that the deleted vertices must be in a total order sequence to create edges of cost $|V| - 1$. This sequence creates a path that includes all but two vertices. Moreover, this path is simple, as deleting a vertex makes it impossible to consider a vertex more than once. Finally, since the cost of edges are at $|V| - 1$, the remaining two vertices are source and terminating vertices of a Hamiltonian path.

5.3 Optimal Longest Sampling Period

In order to cope with the exponential complexity of our optimization problem, we transform it into *Integer Linear Programming* (ILP). ILP is a well-studied optimization problem and there exist numerous efficient ILP solvers. The problem is of the form:

$$\begin{cases} \text{Minimize} & c.\mathbf{z} \\ \\ \text{Subject to} & A.\mathbf{z} \geq \mathbf{b} \end{cases}$$

where A (a rational $m \times n$ matrix), c (a rational n-vector), and $\mathbf{b}$ (a rational m-vector) are given, and, $\mathbf{z}$ is an n-vector of integers to be determined. In other words, we try to find the minimum of a linear function over a feasible set defined by a finite number of linear constraints. It can be shown that a problem with linear equalities and inequalities can always be put in the above form, implying that this formulation is more general than it might look.

5.3.1 Mapping to Integer Linear Programming

We now describe how we map the optimization problem to ILP. Our mapping takes the critical control-flow graph $CFG = \langle V, v^0, A, w \rangle$ of the inspected program and a desired sampling period SP as input. Our objective is to find the minimum number of vertices that must be removed from V through transformation IT.

Integer variables. Our ILP model employs the following sets of variables:

1. $\mathbf{x} = \{x_v \mid v \in V\}$, where each x_v is a binary integer variable: if $x_v = 1$, then vertex v is removed from V, whereas $x_v = 0$ means that v remains in V.
2. $\mathbf{a} = \{a_{v_{(i)}} \mid v \in V \ \wedge \ 0 < i \leq \text{outdegree of } v\}$: where each $a_{v_{(i)}}$ is an integer variable which represents the weight of a unique arc originating from vertex v (i.e., no two $a_{v_{(i)}}$ represent the weight of the same outgoing arc). This variable is needed to store the new weight of an arc created by merging a sequence of arcs. For example, in Figure 4.3(b), initially, variable $a_{B_{2_{(1)}}} = 1$. However, if $x_{B_3} = 1$ (i.e., vertex B_3 is collapsed), then $a_{B_{2_{(1)}}} = 3$.
3. $\mathbf{y} = \{y_{v_{(i)}}, y'_{v_{(i)}} \mid v \in V \wedge 0 < i \leq \text{outdegree of } v\}$, called *choice variables*, where each $y_{v_{(i)}}$ and $y'_{v_{(i)}}$ is an integer variable. The application of this set is described later in this section.

Constraints for the initial basic block. Since we always want a sample at the beginning of the program to extract the initial value of variables of interest, we add the following constraints:

$$x_{v^0} = 0 \tag{5.1}$$

$$a_{v^0_{(i)}} = w(v^0, u) \tag{5.2}$$

where $0 < i \leq$ outdegree of v^0 and (v^0, u) is an arc in A. Note that for each outgoing arc from v^0, the ILP model will have Constraint 5.2.

Constraints for arc weights and internal vertices. Since our goal is to ensure that the weight of all arcs become at least SP, if there exists an arc of weight less than SP, then the target vertex of the arc must be removed from the graph. Thus, for every arc $(u, v) \in A$, we add the following constraint:

$$a_{u_{(i)}} + SP.x_v \geq SP \tag{5.3}$$

where $a_{u_{(i)}}$ represents arc (u, v).

Next, we add constraints for calculating the new weights of arcs when vertices are deleted from CFG. We distinguish two cases:

- **Case 1:** If $x_v = 0$, for some $v \in V$, then for each $a_{v_{(i)}}$, where $0 < i \leq$ outdegree of v, $a_{v_{(i)}}$ represents the weight of a unique arc originating from vertex v.
- **Case 2:** If $x_v = 1$, then for each $a_{v_{(i)}}$, where $0 < i \leq$ outdegree of v, $a_{v_{(i)}} = a_{v_{(i)}} + a_{u_{(j)}}$, where $a_{u_{(j)}}$ represents the weight of the arc $(u, v) \in A$. Note that in this case, although vertex v is removed, for simplicity, we use variables $a_{v_{(i)}}$ as the weight of the newly created arcs. Also note that in this case, outgoing arcs from u automatically satisfy Constraint 5.3.

In order to make these cases mutually exclusive in ILP, we use the choice variables with the following properties:

- **Prop. 1:** The values of $y_{v_{(i)}}$ and $y'_{v_{(i)}}$ are such that one of them is zero and the other is $a_{u_{(j)}}$. This property enforces mutual exclusiveness of the above cases.
- **Prop. 2:** If $x_v = 1$, then for all i, $0 < i \leq$ outdegree of v, $y_{v_{(i)}} = a_{u_{(j)}}$ and $y'_{v_{(i)}} = 0$. On the contrary, if $x_v = 0$, then $y_{v_{(i)}} = 0$ and $y'_{v_{(i)}} = a_{u_{(j)}}$.

In order to enforce Prop. 1, we use a special data structure implemented in our ILP solver called *Special Ordered Set Type 1* (sos_1), where at most one variable in a set can take a

positive value while all other variables in the set must have a value of zero. The following constraints enforce Prop. 1 and 2:

$$y_{v_{(i)}} + y'_{v_{(i)}} = a_{u_{(j)}} \tag{5.4}$$

$$sos_1(y_{v_{(i)}}, y'_{v_{(i)}}) \tag{5.5}$$

$$1 \leq x_v + y'_{v_{(i)}} \leq a_{u_{(j)}} \tag{5.6}$$

Note that Constraints 5.4-5.6 are duplicated for all i, $0 < i \leq$ outdegree of v. The following constraints implement Cases 1 and 2, respectively. These constraints target variable $a_{v_{(i)}}$ which represents an arc $(v, v') \in A$.

$$w(v, v') + a_{u_{(j)}} - y'_{v_{(i)}} = a_{v_{(i)}} \tag{5.7}$$

$$y_{v_{(i)}} + w(v, v') = a_{v_{(i)}} \tag{5.8}$$

For example, if v is deleted (i.e., $x_v = 1$), then we have $y_{v_{(i)}} = a_{u_{(j)}}$ and $y'_{v_{(i)}} = 0$ by Constraints 5.4-5.6. Moreover, when v is deleted, the weight of the newly created arc $a_{v_{(i)}}$ will be $a_u + w(v)$. This is ensured by Constraints 5.7 and 5.8. Note that Constraints 5.7 and 5.8 are duplicated for all i, $0 < i \leq$ outdegree of v.

Now, we duplicate Constraints 5.4-5.8 for each incoming arc to vertex v. More specifically, for arcs $(u_1, v), (u_2, v) \cdots (u_n, v)$, we instantiate Constraints 5.4-5.8 by further duplicating each variable $a_{v_{(i)}}$ to $a^{u_1}_{v_{(i)}}, a^{u_2}_{v_{(i)}} \cdots a^{u_n}_{v_{(i)}}$. We note that existence of multiple incoming arcs in a control-flow graph is due to the existence of conditional and *goto* statements in the input program. Since the depth of nested conditional statements is not normally high, we do not expect to encounter an explosion in the number of a-variables in our ILP model.

Handling loops. Recall that we argued that vertices with self-loops cannot be removed. Self-loops are created when we apply the *IT* transformation on vertices of a cycle in a critical control-flow graph. To ensure that self-loops are not removed, we add a constraint to our ILP model, such that from each cycle $v_1 \to v_2 \to \cdots \to v_n \to v_1$, only $n-1$ vertices can be deleted:

$$\sum_{i=1}^{n} x_{v_i} \leq n - 1 \tag{5.9}$$

We note that cycles can be identified when we construct *CFG* and there is no need for graph exploration to enumerate them.

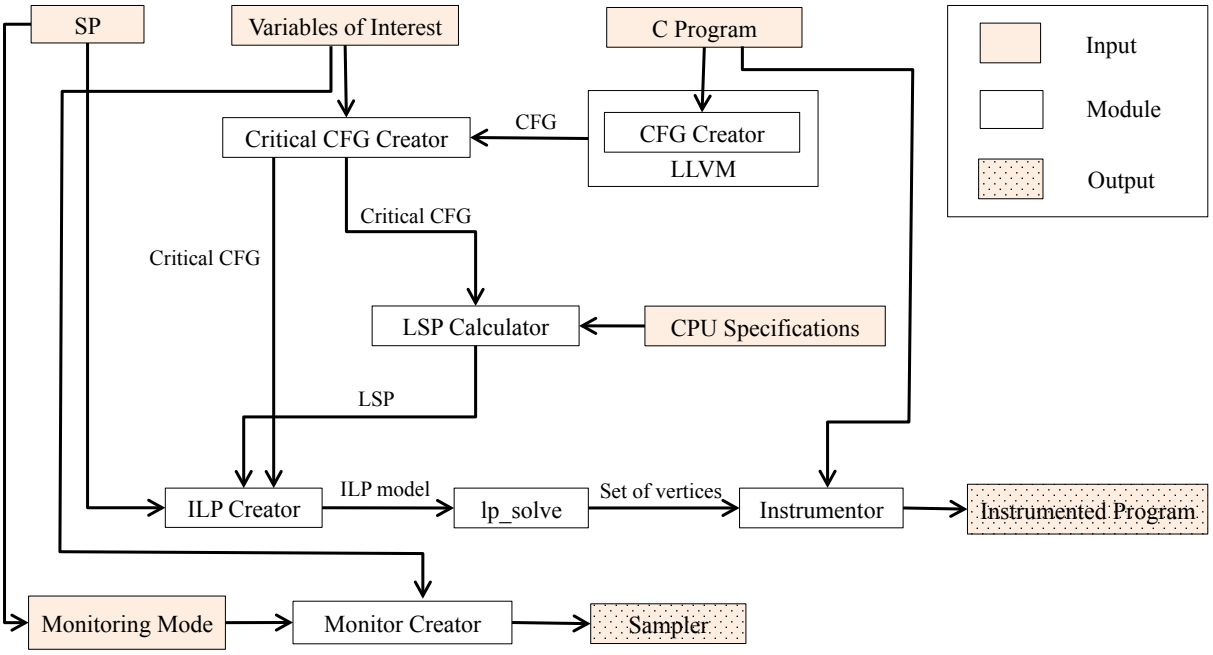


Figure 5.2: Tool chain

Objective function. Finally, we state our objective function, where we aim at minimizing the set of vertices removed from *CFG*:

$$\text{Minimize} \sum_{v \in V} x_v \tag{5.10}$$

5.3.2 Experimental Settings

In this section, we present our tool chain and the experimental configurations.

Implementation and Tool Chain

Figure 5.2 shows our tool chain. Our tool chain consists of four main phases:

- The *CFG* phase which is responsible with creating the critical control-flow graph of the program.

- The *LSP* calculation phase which is responsible with finding the longest sampling period.
- The ILP phase which is responsible for creating the ILP model and solving the optimization problem to find the minimum number of critical vertexes which need to be collapsed.
- The monitoring phase which is responsible with incorporating the sampler process of the time-triggered monitor which conducts the monitoring of the program.

We note that at this point in our experiments, we are not concerned with actual verification of LTL_3 properties. Our main objective is to study different aspects of the monitoring overhead for data extraction only (i.e., the sampler). The actual verification at runtime can be done using the tools introduced in [9] [12].

The *CFG* phase contains two main components, the *CFG* creator and the critical *CFG* creator. The *CFG* creator is implemented over LLVM [64]. It takes the source code of the inspected program as input and produces the program's control-flow graph. Note that each vertex of the control-flow graph includes its best-case execution time and the line number of instructions incorporated within the vertex. The critical *CFG* creator is a Java application which receives the control-flow graph from the *CFG* creator and the list of variables of interest (i.e., variables in $\mathcal{V}_\Pi$) from the user. As a result, the critical *CFG* creator creates the critical control-flow graph and provides it to the *LSP* calculator phase.

The *LSP* calculator phase contains a Java application which receives the critical control-flow graph of the program from the critical *CFG* creator and calculates the longest sampling period of the program in the form of CPU cycles. The *LSP* calculator can also calculate the sampling period in the form of nano-seconds, if the user provides CPU specifications.

The ILP phase contains three main components, the ILP creator, ILP solver, and the instrumentor. The ILP creator is a Java application which receives the critical control flow graph from the critical *CFG* creator, the longest sampling period from the *LSP* calculator, and the intended sampling period SP. The ILP creator returns an ILP model within the format acceptable by the ILP solver. Our ILP solver is the mixed integer linear programming (MILP) solver, lp_solve [70]. lp_solve receives the ILP model and solves the ILP problem. As a result lp_solve returns the set of critical vertices which need to be collapsed. The instrumentor receives the set of collapsed critical vertices from lp_solve and finds the instructions in the program's source code that are incorporated within the collapsed critical vertices. Then, the instrumentor creates a duplicate of the program's source code and instruments the copied source code appropriately. In other words, after

each instruction within a collapsed vertex, the instrumentor adds an instruction which stores the variable of interest updated by the instruction within the history. In the end, the instrumentor returns an instrumented copy of the program's source code.

The monitoring phase contains a Java application (monitor creator) which creates the sampler process. The sampler process is a C program which runs in parallel with the inspected program. The monitoring phase can create one of the following three monitoring modes for the sampler process:

- **Event-triggered (ET monitor):** The program execution halts when an instruction changes the value of a variable of interest (i.e., a critical instruction is executed) and invokes the sampler. The sampler reads the new value of the variable of interest and informs the program to resume execution.
- **Time-triggered with no history (TT monitor):** The sampler is time-triggered. The sampler has a timer that represents the time-triggered monitor's sampling period. Hence, the sampler sets this timer to the sampling period calculated by the *LSP* calculator. When the timer goes off, the sampler halts the execution of the program and reads the value of all variables of interest. Then, the sampler resets its timer and informs the program to resume execution.
- **Time-triggered with history (TTH monitor):** This setting incorporates our ILP optimization. The monitoring is performed over the instrumented copy of the source code that is provided by the instrumentor. The sampler functions as of the TT monitor. In this setting, the sampler sets its timer to the intended sampling period (i.e., SP) provided by the user, and it also extracts the data in the history (i.e., auxiliary memory), in addition to the variables of interest.

In addition, the monitor creator receives the variables of interest and provides read access for these variables to the sampler process.

Experimental Configurations

Our case studies are from the SNU [2] benchmark suite. The experimental setting is as follows. In each program, the main function runs 500 times, where at each iteration the main function receives new input values from the environment. For each program, we find the top two variables which play the most part in the program's runtime behavior. In other words, we find the two variables which are most used by the program's instructions.

Then, we consider these two variables as the variables of interest (i.e., variables in $\mathcal{V}_\Pi$). The program and the time-triggered monitor run on an MCB1700 board with the RTX real-time operating system.

To evaluate the TT and TTH monitors, we consider the following metrics:

1. The execution time of the monitored program. This value projects the overall amount of monitoring overhead at run time.
2. The absolute jitter (i.e., the difference between the minimum and maximum value) of the overhead of the monitor invocations (i.e., monitoring samples) throughout the program run. This metric is of importance, since when the absolute jitter of the overhead of the monitor invocations is smaller, the monitor has more predictable behaviour.
3. The amount of memory used by the TTH monitor.

5.3.3 Experimental Results

In this section, we describe the results of our experiments. In particular, we analyze:

1. The monitoring overhead of different monitors (i.e., ET, TT, TTH).
2. The impact of employing different monitors on memory usage and history size.

Monitoring Overhead at Run Time

We analyze the runtime monitoring overhead based on the *absolute* jitter of the monitor invocations, the actual execution time of the monitored program, and redundant sampling.

Absolute Jitter Analysis:

Notice that each overhead of a monitoring invocation (MIO) at run time is caused by:

1. the overhead of interrupting/resuming the program execution and invoking the sampler, and
2. reading the values of the variables of interest as well as the variables stored in the history (i.e., auxiliary memory).

Recall that one of the main goals of designing a time-triggered monitor is to obtain *bounded monitoring overhead* at run time. In other words, we hypothesize that the absolute jitter of MIO of a TT monitor is less than the absolute jitter of MIO of an ET monitor. To validate our hypothesis, consider Figures 5.3(a), 5.3(b), 5.3(c), and 5.3(d). These figures show MIO of the three monitoring modes throughout the program run of bs, qsort, select, and sqrt, respectively. Note that, in these figures $50 * TT$ and $100 * TT$ represent the TTH monitor where the longest sampling period has been increased by 50 and 100 times respectively. In general, the ET monitor invocations are irregularly distributed throughout the program execution. For instance, Figure 5.3(a) shows that program bs has an execution path which does not incorporate any critical instructions, and hence, the ET monitor is not invoked at all (the execution between 1500μs and 2000μs). Moreover, recall that the overhead caused by each invocation is proportional to the type of the variable of interest read by the monitor. Hence, MIO may vary considerably from one invocation to another when the type of variables of interest vary. This, in turn, results in a large absolute jitter for the MIO (e.g., qsort in Figure 5.3(b)). Thus, the ET monitor introduces probe-effects, which in turn may create unpredictable and even incorrect behavior from the monitored program. This anomaly is, in particular, unacceptable for real-time embedded and mission-critical systems.

On the contrary, the TT monitor invocations are evenly distributed throughout the program execution. In addition, since the number and type of variables of interest read at each sample remains constant, MIO is not subject to any bursts. Recall that the TT monitor reads all the variables of interest at each sample. Hence, the absolute overhead remains consistent and bounded which results in a small absolute jitter for MIO (see Figures 5.3(a), 5.3(b), 5.3(c), and 5.3(d)). Thus, the TT monitor exhibits predictable monitoring and bounded monitoring overhead required for runtime verification of real-time embedded systems (see Chapter 3). Consequently, the monitored program exhibits a predictable behaviour. As can be seen in Figure 5.3, the TT monitor may potentially impose larger overhead compared to the ET monitor, which as a result, extends the overall execution time of the monitored program. Nonetheless, in many commonly considered embedded applications, designers prefer predictability at the cost of larger overhead.

For qsort, Figure 5.3(b) shows that the absolute jitter of MIO of the TT monitor is less than the absolute jitter of MIO of the ET monitor. In qsort, the variables of interest are of types array of int, int, and long. The different values of MIO of the ET monitor in Figure 5.3(b) shows how different types of variables can affect the value of MIO. For instance, when an entry in the array changes, the ET monitor reads all the variables stored within the array. On the contrary, when the long variable changes, the ET monitor only reads the changed variable. Hence, the overhead of reading variables of interest may vary

significantly from one variable to another. Thus, Figure 5.3(b) is a clear indication of how different types of variables of interest affect the value of MIO. As for bs, Figure 5.3(a) shows that the absolute jitter of MIO of the TT monitor and the absolute jitter of MIO of the ET monitor are approximately equal. For such programs, this is caused by the condition, where all variables of interest are of the same type. For instance in bs, all the variables of interest are of type int, and hence, the ET monitor extracts the same type of variables at each invocation. Thus, the ET monitor simulates the invocation condition of a TT monitor, where the type and number of variables extracted by the monitor are all similar. As a result, the absolute jitter of MIO of the ET monitor is similar to the absolute jitter of MIO of the TT monitor. Observations regarding qsort stand for select and sqrt (See Figures 5.3(c) and 5.3(d)).

The situation is more complex for TTH monitors. A TTH monitor also reads the variables of interest stored in the history at each monitoring sample. Hence, the MIO of a TTH monitor depends on the overhead of reading the history as well. Since the number and type of variables stored in the history can differ from one sample to another, the absolute jitter of MIO may not be smaller than the absolute jitter of MIO of the ET monitor. For qsort, select, and sqrt, the absolute jitter of MIO of the TTH monitor with the intended sampling period of both $50 * TT$ and $100 * TT$ is less than the absolute jitter of MIO of the ET monitor (see Figure 5.4(a)). This is not the case in all programs, for instance, in program jfdctint, the absolute jitter of MIO of the TTH monitor with the intended sampling period of both $50 * TT$ and $100 * TT$ is greater than the absolute jitter of MIO of the ET monitor. Our deeper analysis shows that this is caused by a large difference between the number of *instrumentation* instructions executed between two samples. Recall from Subsection 5.3 that we use ILP to find the set of critical instructions to collapse as to achieve the intended sampling period SP, and as a result, the variables updated by these instructions are stored within the history. Hence, immediately after each of these critical instructions, the Instrumentor module in our tool chain adds an instruction which stores such updated variables into the history. We refer to these instructions as *instrumentation* instructions. Since our ILP model only focuses on finding the solution which incorporates the minimum amount of history, the instrumentation instructions may be unevenly distributed throughout the program run. Thus, there is a possibility that the number of executed instrumentations between two samples vary.

The fluctuation in the number of instrumentations executed between two samples causes an increase in the absolute jitter of the MIO. Figure 5.4(b) shows the absolute jitter of the executed instrumentations between two consecutive samples. It is clear that a larger absolute jitter for the executed instrumentation causes a larger absolute jitter for MIO. As can be seen, for jfdctint, the absolute jitter of the executed instrumentation to achieve the

intended sampling period of $50 * TT$, is 25, which is a large value in comparison to the other SNU programs.

Figures 5.3(a) and 5.3(b) also show that in each monitoring mode, the execution time of the program changes (Note that the x-coordinate within the figures presents the execution time of the programs). The execution time of a monitored program depends on:

1. The MIO of the monitor.
2. The number of monitor invocations.

Recall that at each monitor invocation, the sampler process stops the program execution and resumes its execution when the sampler finishes reading the variables of interest. Hence, the time the program *resumes* its execution depends on the MIO. For instance, the MIO of the TT monitor is larger than the ET monitor. In addition, the TT monitor intervenes with the program more often. As a result, the execution time of bs, qsort, select, and sqrt monitored with the TT monitor is longer than their execution time when monitored with the ET monitor. We will discuss the characteristics of a monitored program's execution time in more detail in the next subsection.

Execution Time and Redundant Sampling Analysis:

As for the effect of the monitoring overhead on program execution, Figures 5.5(a) and 5.5(b) show the execution time of the programs of SNU while being monitored with our three monitoring modes. The results show that the execution time of a program monitored with the TT monitor is larger than the execution time of the program monitored with the ET monitor. This excessive overhead is caused by the following characteristics of the TT monitor:

- The monitor invocation happens more often in the TT monitor compared to the ET monitor.
- The MIO of the TT monitor is larger compared to the ET monitor. This is caused by the fact that at each invocation, the TT monitor reads all the variables of interest from the program while the ET monitor reads only one variable of interest.

The side effect of high volume monitor invocation is *redundant sampling*. A redundant sample is when the monitor takes a sample while the program has not executed a critical instruction since the last monitoring sample. The TT bar in Figures 5.6(a) and 5.6(b)

shows the number of redundant samples taken by the TT monitor at run time, and the *Event* bar shows the number of critical instructions executed at run time. The ratio of redundant samples to the number of executed critical instructions is the metric which defines the excessive overhead of the TT monitor. We refer to this ratio as the *redundant sample ratio*. Thus, a larger redundant sample ratio results in larger excessive monitoring overhead, and hence, a longer execution time for the monitored program.

In a program such as minevr, the monitor takes 209,213 redundant samples which results in a redundant sample ratio of 0.70, which is a considerably large value compared to the other SNU programs. Consequently, the execution time of minevr when monitored with the TT monitor is 1.33 times longer than its execution time when monitored with the ET monitor (see Figure 5.5(b)). Figure 5.6(c) shows the average frequency in which the monitor takes a redundant sample. Clearly, a larger redundant sample ratio results in a higher frequency of redundant samples. Hence, it is desirable to decrease the frequency of redundant samples. To this end, we use history to increase the longest sampling period, and hence, decrease the frequency of redundant samples. In the SNU programs, by using history to achieve the intended sampling periods of $50 * TT$ and $100 * TT$, the number of redundant samples reduces to zero. In other words, in the SNU programs, the intended sampling periods of $50 * TT$ and $100 * TT$ do not result in redundant samples, meaning that when the TT monitor takes a sample, the program has executed at least one critical instruction. Note that this may not be the case for programs other than the SNU programs. In other words, by using history, the redundant samples of the TT monitor does not reduce to zero for all possible programs.

In general, the monitoring overhead imposed by the TTH monitor is less than the TT monitor. Recall that the MIO is twofold. Our studies show that the overhead imposed by stopping/resuming the program execution and invoking the sampler makes up the majority of MIO. Consequently, when the TTH monitor increases the sampling period, it also reduces the number of monitor invocations (i.e., monitoring samples). As a result, the overhead imposed by the TTH monitor is less than the TT monitor. Figures 5.5(a) and 5.5(b) show the reduction in the execution time of the programs when using the TTH monitor compared to using the TT monitor.

In some programs such as sqrt, select, and qurt, the execution times of the programs monitored with the TTH monitor with sampling period of $50 * TT$ and $100 * TT$ are less than the execution time of the programs when monitored with the ET monitor. Our studies show that in such programs, by using history, the number of monitor invocations reduces by more than 50% (e.g., in sqrt the reduction is 93%). Figures 5.5(c) and 5.5(d) show the number of monitor invocations for each monitoring mode. These figures show that the number of monitor invocations of the TTH monitor for both $50 * TT$ and $100 * TT$

are less than the monitor invocations of the ET monitor. These figures show that the TTH monitor does not introduce redundant samples, and hence, does not impose excessive and redundant overhead. Also, our studies show that in these programs, the overhead of stopping/resuming the program execution and invoking the monitor still makes up the majority of the MIO. Hence, the effect of the reduction in the number of invocations overcomes the effect of the increase in the overhead of reading the values of the variables of interest (Recall that the TTH monitor must also read the variables of interest stored in the history). Thus, the monitoring overhead imposed by the TTH monitor becomes less than the overhead of the ET monitor. To this end, TTH monitors provide efficient monitoring which is required for a runtime verification framework of real-time embedded systems.

On the other hand, in programs such as lms and insertsort, the execution time of the programs monitored with the TTH monitor with sampling period of $50 * TT$ is more than their execution time when monitored with the ET monitor. In these programs, our studies show that the overhead of reading the variables of interest exceeds the overhead of stopping/resuming the program execution and invoking the monitor. Hence, the effect of the increase in the overhead of reading the values of the variables of interest overcomes the effect of the decrease in the number of monitor invocations. Thus, the overhead imposed by the TTH monitor is larger than the overhead of the ET monitor.

History Size at Run Time:

Regarding the TTH monitor, recall that we prohibited deletion of self-loops from critical control-flow graphs. Hence, if some critical instructions reside in loop structures, the minimum sampling period of the loop structures can determine the longest sampling period. For the SNU programs, the majority of the critical instructions reside in loops, and hence, in such a situation, employing history does not result in a considerable increase in the sampling period (e.g., for fibcall the sampling period does not increase at all). To overcome this problem, we use profiling to estimate the upper bound of the number of times each loop structure takes for each program. We leverage gcov to carry out the profiling. With respect to the upper bound on the loops and type of variables of interest updated within the loops, we devise a size for the memory location of the history (i.e., auxiliary memory). For instance, if a loop structure runs at most 100 times and within it a variable of interest of type integer is updated, we devise a memory location of at least the size $int\_size * 100$ for the history. In addition, we note that solving the corresponding ILP problem for all programs of SNU takes an average of 56 seconds. This clearly shows that we are not even close to the boundaries of ILP solving.

Figure 5.7(a) shows the average number of instrumentations executed between two consecutive samples. In other words, it shows the average number of data stored in the history between samples. Figure 5.7(b) shows the average amount of memory consumed by the history in between two consecutive samples. Note that the amount of history consumption depends on the number and type of variables stored within the history. The encouraging outcome from the experimental results shown in Figure 5.7(b) is that with a small amount of additional memory, we can severely increase (e.g., by 50 and 100 times) the sampling period of the TT monitor. For instance, program lms uses the most amount of extra memory (5,088 bits) to increase its sampling period by a factor of 100. Hence, results show that by using approximately 5kbits of memory, the execution time of lms decreases by 57% (see Figure 5.5(b)). Thus, the experimental results encourage the use of TTH monitors.

Resource Management:

Although the relative data may seem to indicate that event-triggered approaches use less resources than time-triggered approaches, this is incorrect. Real-time applications must operate even under worst-case scenarios and as such, the worst-case behavior is of interested instead of the average case behavior.

To demonstrate this, let us consider the program sqrt and the associated measurements. Figure 5.8(a) shows the cumulative overhead for the program. The x-axis show the execution time of the application in seconds and the y-axis shows the overhead up to that execution time. It clearly shows that the event-triggered system has less overhead than the time-triggered approaches. However, this also depends on the worst-case behaviour.

Figure 5.8(b) shows the upper bound on the number of events (i.e., monitor invocations) for different durations. The x-axis shows the length of the duration in which events occurred. The y-axis shows the maximum number of events (i.e., monitor invocations) found in at least one time interval of the given duration. For example, the time-triggered approach resulted in an upper bound of 5,000 events in at least one observed interval of 250 seconds. The interesting point is that the event-triggered approach consistently has a higher upper bound on the number of events (i.e., monitor invocations) for any given duration. This means that for a real-time application, in which developers have to reserve resource budgets to ensure the timeliness of the system, the event-triggered approach will require a larger reservation of resources than time-triggered approaches. This is due to, in the worst case, more events occurring and the need to reserve resources for the worst case.

The final decision on which scheme requires the lower resource partition depends on several factors such as interruption overhead, context-switch overhead, messaging overhead,

etc. Thus Figure 5.8(b) reports the value on the y-axis as the number of observed events. However, this leaves the basic argument intact. Thus, it can be concluded, that TT and TTH monitors reduce over-provisioning of resources which is required to runtime verify real-time embedded systems.

5.4 Near-Optimal Longest Sampling Period

The transformation to an ILP enables us to achieve a desired sampling period SP while using the minimum number of instrumentation instructions to store updates in variables in $\mathcal{V}_\Pi$. It is possible to solve the corresponding ILP for moderate size programs (such as MiBench and SNU programs), but for larger programs, the exponential complexity poses a serious problem. With this motivation, we focused on developing polynomial-time algorithms that find near-optimal solutions to our optimization problem. Our algorithms are inspired by an observation made in Subsection 5.3.3. By comparing the reduction in monitoring overhead by using history (see Figures 5.5(a) and 5.5(b)) with the increase in the memory consumption (see Figure 5.7(b)) of SNU programs, we realized that even with increasing the sampling period by a factor of 100, the memory usage increases at most by 4%. Our experiments on other programs such as the MiBench bench suite [1] exhibit the same behaviour. Hence, we believe that the impact of increasing the longest sampling period on memory usage is negligible. This observation suggests that near-optimal solutions to the optimization problem are likely to be sufficiently effective.

We propose three polynomial-time heuristics. All heuristics are over-approximations and hence, sound (i.e., they do not cause overlooking of critical instructions). In the remainder of this section, we present the algorithms along with the experimental results in detail [85] [104]. In general, the heuristics take a critical control-flow graph CFG and a desired sampling period SP as input and return a set U of vertices to be collapsed from CFG using the transformation $IT(CFG, v)$ (see Section 5.1) to achieve SP.

5.4.1 Heuristic 1: The Greedy Heuristic

Our first heuristic is a simple greedy algorithm (see Algorithm 1):

1. It removes all the vertices in CFG where the weights of all its incoming and outgoing arcs are greater than or equal to SP (Line 2) . Obviously, these vertices do not take part in the process of achieving a critical control-flow graph with the minimum arc

weight of at least SP, since their incoming and outgoing edges already have a weight larger than SP.

2. It explores CFG to find the vertex incident to the maximum number of incoming and outgoing arcs whose weights are strictly less than SP (Line 4). Our intuition is that collapsing (i.e., deleting) such a vertex results in removing a high number of arcs whose weights are less than SP.

3. It collapses vertex v identified on Line 4. This operation (Line 5) results in merging incoming arcs to v with outgoing arcs from v in the same fashion as transformation $T(CFG, v)$.

4. Since basic block v contains a critical instruction, we need to apply transformation $IT(CFG, v)$ to store the updated value of the variable in $\mathcal{V}_\Pi$ at v. Thus, it adds v to U (Line 6).

5. It repeats Lines 3-7 until the minimum arc weight of CFG is greater than or equal to SP (the while-loop condition in Line 3).

6. If all the vertices of CFG are collapsed, then regarding the program's structure, it can not achieve SP. Thus the algorithm declares failure.

5.4.2 Heuristic 2: The Vertex Cover Heuristic

The second heuristic is based on a solution to the *minimum vertex cover* problem: Given a (directed or undirected) graph $G = \langle V, E \rangle$, our goal is to find the minimum set $U \subseteq V$, such that each edge in E is incident to at least one vertex in U. The minimum vertex cover problem is NP-complete, but there exist several approximation algorithms that find near-optimal solutions (e.g., the 2-approximation in [30]).

Assuming that the critical control-flow graph CFG is the graph at hand, our algorithm (see Algorithm 2) works as follows:

1. It removes all the vertices in CFG where the weights of all its incoming and outgoing arcs are greater than or equal to SP (Line 2).

2. It computes an approximate vertex cover of graph CFG (Line 4), denoted as vc. Our intuition is that since the graph is pruned (Step 1) and the vertex cover vc covers

all arcs of CFG, collapsing all vertices in vc may result in removing all arcs whose weights are strictly less than SP. We note that the approximation algorithm in [30] is a non-deterministic randomized algorithm and may produce different covers for the same input graph. As a result, to improve our solution, we run Line 4 multiple times and select the smallest vertex cover. This is abstracted away from the pseudo-code.

3. Similar to Heuristic 1, it collapses each vertex $v \in vc$ (Lines 5-7). This (Lines 5-7) results in merging incoming arcs to v with outgoing arcs from v in the same fashion as transformation $T(CFG, v)$. Since we need to apply transformation $IT(CFG, v)$ to store the updated value of the variable in $\mathcal{V}_\Pi$ at v, we add v to U (Line 7).

4. It repeats Lines 3-8 until the minimum arc weights of CFG are greater than or equal to SP (the while-loop condition in Line 3).

5. If all the vertices of CFG are collapsed, then regarding the program's structure, it can not achieve SP. Thus the algorithm declares failure.

5.4.3 Heuristic 3: The Genetic Algorithm Heuristic

The third heuristic is a genetic algorithm (GA). Our genetic model, takes a desirable sampling period SP and a critical control-flow graph $CFG = \langle V, v^0, A, w \rangle$ as input. As a result, it returns the set of critical vertices which need to be collapsed from CFG to achieve SP. Our genetic model is as follows and we will describe it in detail in the following:

1. *Chromosomes*: Each chromosome represents the list of vertices V. Each vertex in a chromosome is flagged by either the value true or false. The value true represents the condition where the vertex has been chosen to be collapsed.

2. *Fitness Function*: The fitness function of a chromosome is the number of collapsed vertices represented by the chromosome.

3. *Reproduction*: To create a new generation of chromosomes, we use both mutation and crossover.

4. *Termination*: The genetic algorithm terminates when it reaches the upper limit on creating new generations.

The Chromosomes:

Each chromosome in the genetic model has a static length of $|V|$. Each entry of the chromosome is a tuple $\langle \text{vertex id}, \text{min-SP}, \text{value} \rangle$ that represents a vertex in V. *Vertex id* is the vertex identifier, *min-SP* is the minimum weight of the incoming and outgoing arcs of the vertex and *value* indicates whether the vertex has been chosen to be collapsed. If $value = true$ for a vertex v, we need to apply transformation $T(CFG, v)$. We put a restriction on each chromosome that the longest sampling period of the control-flow graph resulting from the collapsed vertices represented by the chromosome must always be at least SP. We refer to the longest sampling period of the resulting control-flow graph as the *chromosome's sampling period*.

Upon initialization, we first set $|\mathcal{G}|$ as the number of chromosomes in each generation in the genetic model. Second, the model randomly creates $|\mathcal{G}|$ chromosomes for the initial generation. To create a chromosome, the model randomly collapses a set of vertices which results in a control-flow graph with a longest sampling period of at least SP. The procedure is as follows:

1. It finds all the vertices in CFG with *min-SP* less than SP and puts them in a set SV.
2. It randomly chooses a vertex $v \in SV$ and collapses v from CFG (i.e., $value = true$) and produces a new control-flow graph $CFG' = T(CFG, v)$.
3. It calculates the sampling period of CFG'. If the sampling period is less than SP, it returns back to Step 1.

Selection/Fitness Function:

Since we aim at increasing the sampling period to SP with the least number of collapsed vertices, the chromosome with the least number of collapsed vertices is the most fit the chromosome. Thus, we define the fitness function as: $\mathcal{F} = \mathcal{C}_{chr}$, where $\mathcal{C}_{chr}$ is the number of collapsed vertices in chromosome chr.

Reproduction:

The genetic model uses both *mutation* and *crossover* to evolve the current generation into a new generation [77].

For the crossover, the algorithm uses a *one-point* crossover [77] to create new chromosomes for the next generation. The choice of parents is random. The crossover cuts the

two parents into half and creates two children by swapping the halves between the parents. It checks each child to see if its sampling period is at least SP. If so, it adds the child to the set of chromosomes of the next generation; if not, it passes the child to the mutation step.

For the mutation, the algorithm takes the children passed over by the crossover process and evolves each child as follows.

1. It finds all the vertices with *min-SP* less than SP and puts them in a set SV.
2. It randomly chooses a vertex $v \in SV$ to collapse (i.e., $value = true$).
3. It finds the set of collapsed vertices in the child chromosome with *min-SP* larger than SP and puts them into a set PV.
4. It randomly chooses a vertex $u \in PV$ to restore back into the control-flow graph represented by the child chromosome (i.e., $value = false$).
5. It will check if the longest sampling period of the new child chromosome is at least SP. If the longest sampling period is less than SP, it will return to Step 1 and repeat the steps again until the longest sampling period of the child chromosome reaches SP or when it exhausts the limit that we have set for the number of times a chromosome can be mutated.
6. If the mutation process succeeds to create a new child chromosome with a longest sampling period of at least SP, it adds the chromosome to the next generation.

Sometimes the crossover and mutation processes fail to create $|\mathcal{G}|$ number of chromosomes to populate the next generation. This situation can occur when fewer than $|\mathcal{G}|$ number of child chromosomes have a longest sampling period of at least SP. In this case, our genetic model chooses the most fit chromosomes from the current generation and adds them to the next generation to create a population of $|\mathcal{G}|$ number of chromosomes. In the case where this process results in duplicated chromosomes in the next generation, our genetic model discards one of the duplicates and randomly creates new chromosomes as previously described.

Termination:

Two conditions can terminate the process of creating a new generation.

1. When the genetic model finds a chromosome with a longest sampling period of at least SP and has collapsed the same number of vertices as the optimal solution achieved by ILP, or

2. When the genetic algorithm reaches an upper bound on the number of generations.

In the second case, from all the created generations, we choose the chromosome with the lowest fitness value $\mathcal{F}$.

5.4.4 Experimental Results

In this section, we present the experimental results from our heuristics. We use the same experimental setting as in Section 5.3.2 with the following exceptions.

1. We use programs from the MiBench [45] test suite instead of SNU. We made this choice because unlike SNU, MiBench provides larger and more complex programs where our ILP solution is intractable.

2. We augment the *ILP phase* with the implementation of our three heuristics and rename it to the *Optimization phase*.

3. We set the desired sampling period SP to $40 \times LSP$, where LSP is the longest sampling period of the program under scrutiny.

Performance of Heuristics:

Table 5.1 compares the performance of different programs of MiBench which have been instrumented and monitored based on the solutions from the ILP approach (see Section 5.3) and our three heuristic algorithms. The first column shows the size of the critical control-flow graph of programs in terms of the number of vertices. For each approach, we record the time spent to solve the optimization problem (in seconds) and the suboptimal factor (SOF). SOF is defined as $\frac{sol}{opt}$, where sol and opt are the number of vertices collapsed by a heuristic and the ILP approach, respectively.

Results from Table 5.1 show that all three heuristic algorithms perform faster than solving the ILP via the ILP solver lp_solve. On average, Heuristic 1, Heuristic 2, and Heuristic 3 yield in speedups of 200 000, 7 000, and 9, respectively, where the speedup is defined as the ratio between the execution time required to solve the ILP problem and the

	CFG	*ILP*		*Heuristic 1 (Greedy)*		*Heuristic 2 (VC)*		*Heuristic 3 (Genetic Algo.)*	
	Size($\|V\|$)	**time (s)**	**SOF**	**time (s)**	**SOF**	**time (s)**	**SOF**	**time (s)**	**SOF**
Blowfish	177	5316	–	0.0363	7.8	0.8875	8	383	2.5
CRC	13	0.35	–	0.0002	3.5	0.0852	3	0.254	1.5
Dijkstra	48	1808	–	0.0064	1.2	0.1400	1.2	116	1.7
FFT	47	269	–	0.0042	1.7	0.1737	1.8	74	1.1
Patricia	49	2084	–	0.0054	1.4	0.1369	1.6	140	1.5
Rijndael	70	3096	–	0.0060	1.6	0.2557	2.1	370	1.9
SHA	40	124	–	0.0039	2.2	0.1545	2.2	46	1.3
Susan	20 259	∞	–	3 181	N/A	26 211	N/A	923	N/A

Table 5.1: Performance of different optimization techniques.

execution time of the heuristics until they return a result or terminate with a failure. The execution times of Heuristic 2 are based on running Approximate-Vertex-Cover 500 times to cope with the randomized vertex cover algorithm (see Line 4 in Algorithm 2). Table 5.1 shows that for large size programs, such as Susan, with over 1,000 lines of code, the ILP approach is infeasible, although the heuristics are able to generate a solution.

In our case studies, Heuristic 3 produces results that are closer to the optimal solution compared to Heuristics 1 and 2. The spread of the SOFs is small for results achieved by Heuristic 3. For the conducted experiments, the worst SOF for Heuristic 3 is 2.5 (i.e., for Blowfish), which indicates that this solution will collapse 2.5 times more vertices in the critical control-flow graph compared to the ILP approach. With the exception of Blowfish, Heuristics 1 and 2 also perform well, where the SOF ranges from 1.2 to 3.5. Interestingly, considering the case for Susan, experiments show that results from Heuristics 1 and 2 do not suffer with respect to the size of the problem. In Susan, Heuristics 1 and 2 achieve the desired sampling period by deleting 104 and 180 vertices, respectively, while Heuristic 3's solution collapsed 222 vertices. For Susan, the number of vertices being collapsed is approximately 0.5% to 1% of $|V|$, which indicates that the instrumentation overhead should be small. The SOFs for Dijkstra also indicate an anomaly in the overall trend. Therefore, we conclude that the performance of the heuristics depend on the structure of the critical control-flow graph.

Analysis of Instrumentation Overhead:

We also collected the execution times and memory usage of the instrumented benchmark programs during experimentation. Figure 5.9 shows the execution times and memory usage of our eight benchmark programs. Each plot in Figure 5.9 contains the execution times and

memory usage for the unmonitored program, the program monitored by a time-triggered monitor with sampling period of LSP, and the program monitored by a time-triggered monitor with sampling period of $40 \times LSP$ and the instrumentation points indicated by ILP and heuristic solutions inserted in each program.

Based on Figure 5.9, we observe that programs monitored with the time-triggered monitor with sampling period of LSP run slower than the instrumented programs monitored by the time-triggered monitor with sampling period of $40 \times LSP$. This is expected because the time-triggered monitor requires more processing resources when it samples at higher frequencies. The ILP and heuristic solutions on average reduce the monitoring overhead by 25.19% in Blowfish, 44.36% in CRC, 39.65% in Dijkstra, 37.69% in FFT, 28.54% in Patricia, 60% in Rijndael, 70% in Sha, and 87% in Susan.

We also observe that the variation of the execution times of programs instrumented based on the optimal and heuristic solutions (i.e., ILP, Heuristics 1, 2, and 3) are negligible. The average increase in the execution time of programs instrumented by our heuristic algorithms compared to ILP is 14% in Blowfish, 10% in Dijkstra, 3% in FFT, 2% in Patricia, 6% in Rijndael, and 2% in Sha. CRC is an exception since the average execution time did not change. Although CRC's SOF is on average 2.6, our studies show that not only the number of instrumentation, but also the location of the instrumentation takes part in the overall instrumentation overhead. For instance, an instrumentation located inside a loop that iterates 10 times imposes 10 times more overhead compared to an equivalent instrumentation located outside a loop structure. To this end, results from CRC show that in the worst case, although a solution produces less instrumentation points, it does not result in less instrumentation overhead at run time and hence, faster execution time for the program. As for Susan, since ILP is infeasable, we were not able to calculate the increase in the execution time. Based on these results, we can conclude that in our test cases, using suboptimal instrumentation schemes do not greatly affect the execution time of the program as compared to the execution time of optimally instrumented program.

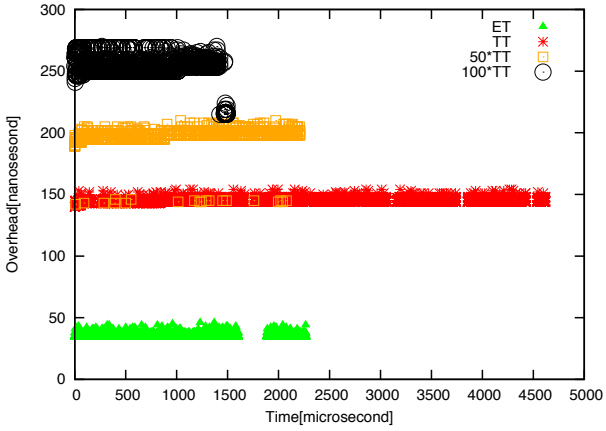
The experimental results from Figure 5.9 show that the increase in memory usage when instrumenting the program based on the results from the heuristics, is negligible during program execution with respect to the program instrumented by the solution produced by ILP, excluding Blowfish. The variation of memory usage for all benchmark programs except for Blowfish generally spans from 0.1 MB to 0.4 MB. Even though the memory usage of Blowfish instrumented with the schemes produced by Heuristic 2 and Heuristic 3 shows an increase of 15 MB of virtual memory, it is still negligible to the amount of memory that is generally available on the embedded systems today. As mentioned before, regarding the set of available experiments, we can not yet conclude which heuristic generally produces the best instrumentation scheme, since we believe that the outcome of each heuristic depends

on the structure of the program's control-flow graph.

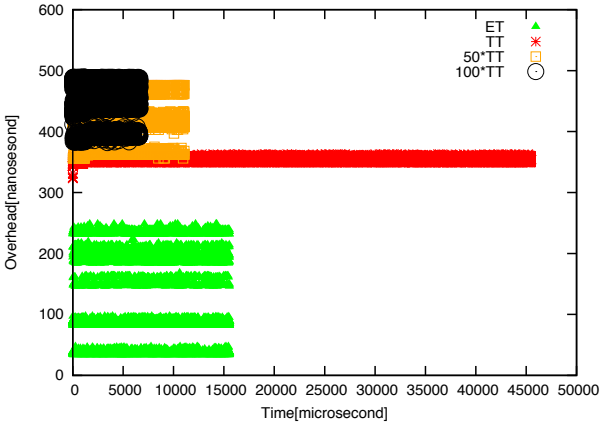
Figure 5.10 shows the percentage increase in the number of instrumentation instructions executed and the percentage increase in the maximum size of history between two consecutive samples with respect to the optimally instrumented benchmark programs (i.e., instrumented with respect to ILP result). Note that logarithmic scales are used in the charts in Figure 5.10. Observe that Susan is not shown in the figure, because solving the optimal solution is infeasible. In most cases, the percentage increase in the number of executed instrumentation instructions and the maximum size of history consumption are below 50%. If we ignore a few more outliers, then most of the percentage increases for both measures are below 20%. We also observe that the percentage increase in the number of executed instrumentation instructions is proportional to the increase in the maximum size of the history consumption between two consecutive samples. This implies that the extra instrumentation instructions (as compared to the optimal solution) are evenly distributed among sampling points.

One may argue that auxiliary memory usage at run time must be in direct relationship with the number of collapsed vertices (i.e., added instrumentation instructions), this is not necessarily true. This is because the number of added instrumentation instructions differs in different execution paths. For example, one execution path may include no instrumentation instruction and another path may include all such instructions. In this case, the first path will build no history and the second will consume the maximum possible auxiliary memory. This is why Table 5.1 shows that Heuristic 3 uses substantially more memory than Heuristic 1 and 2 in Rijndael, although Heuristic 2 collapses more critical vertices. This is also why in Figure 5.10, the amount of auxiliary memory used by a monitored program is not proportional to the number of instrumented critical instructions. Note that in Figure 5.10(a), there is no bar representing CRC for Heuristic 2 and 3. This is because the instrumentation in CRC increases by only 1%. Also in Figure 5.10(b), there is no bar representing CRC for Heuristic 1, 2 and 3. This is because the maximum length of history in CRC increases by only 1%.

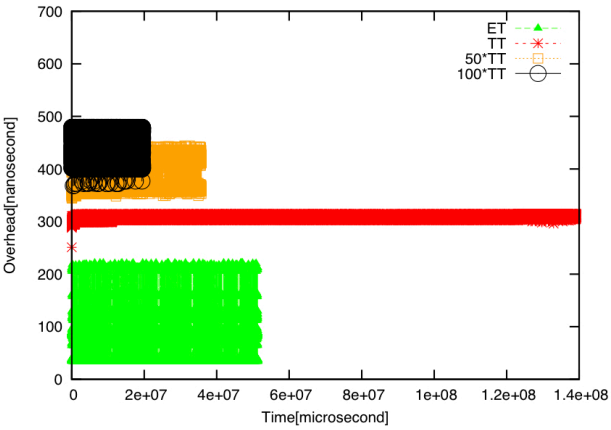
> *We conclude from our experiments that the NP-completeness of the optimization problem is not an obstacle when applying time-triggered monitoring in practice.*



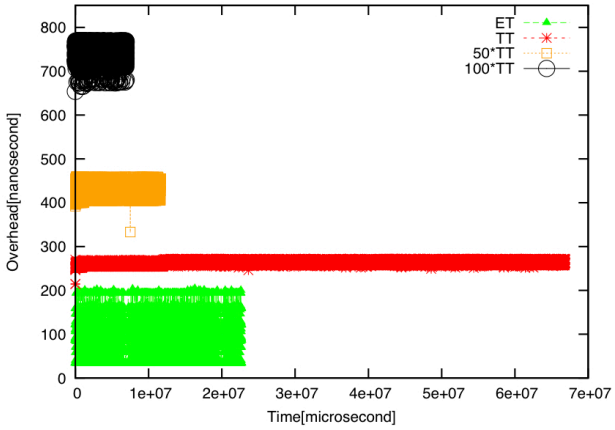
(a) Monitor invocation overhead in bs



(b) Monitor invocation overhead in qsort

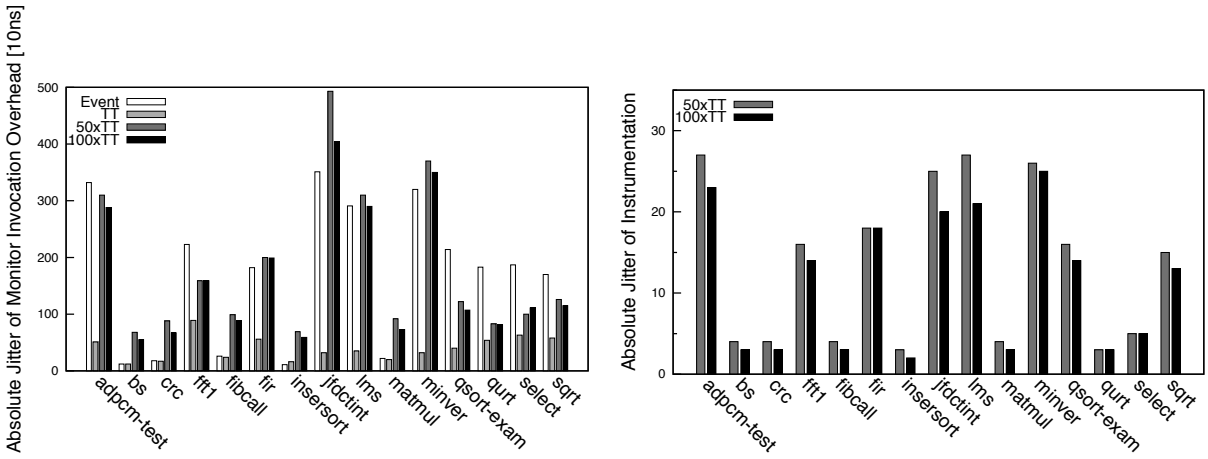


(c) Monitor invocation overhead in select



(d) Monitor invocation overhead in sqrt

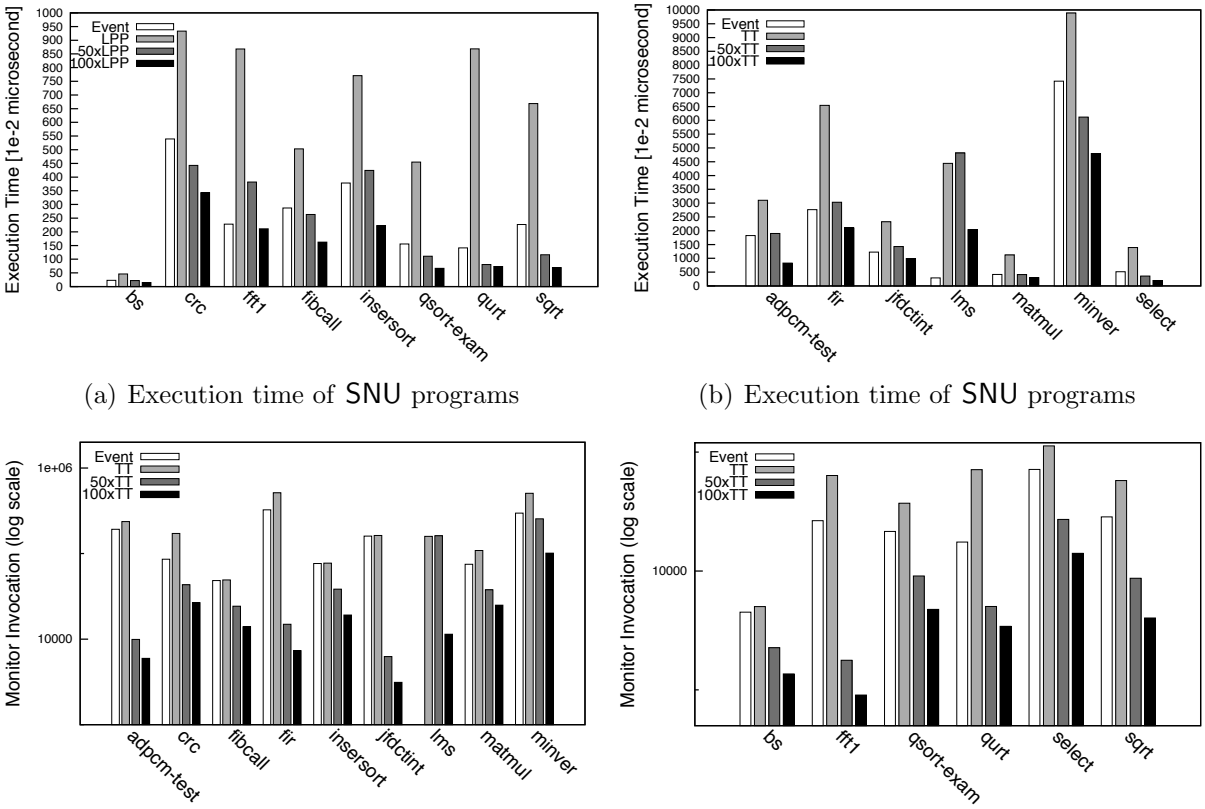
Figure 5.3: Absolute overhead of monitoring invocations



(a) Absolute jitter of monitor invocation overhead

(b) Absolute jitter of instrumentation

Figure 5.4: Absolute jitter of monitor invocation overhead and instrumentation



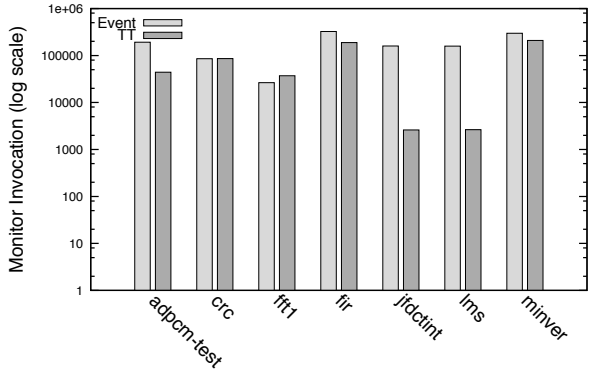
(a) Execution time of SNU programs

(b) Execution time of SNU programs

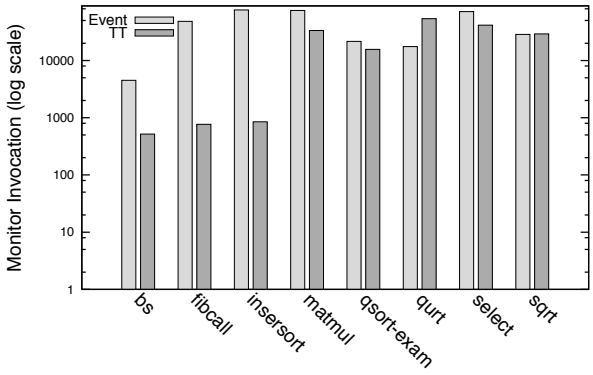
(c) Monitor invocation

(d) Monitor invocation

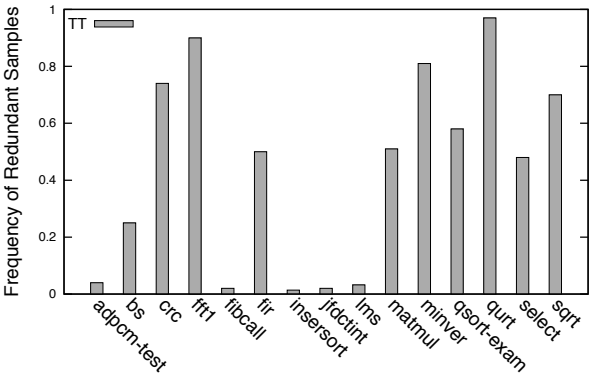
Figure 5.5: Monitoring overhead and monitoring invocation



(a) Redundant samples of TT monitor

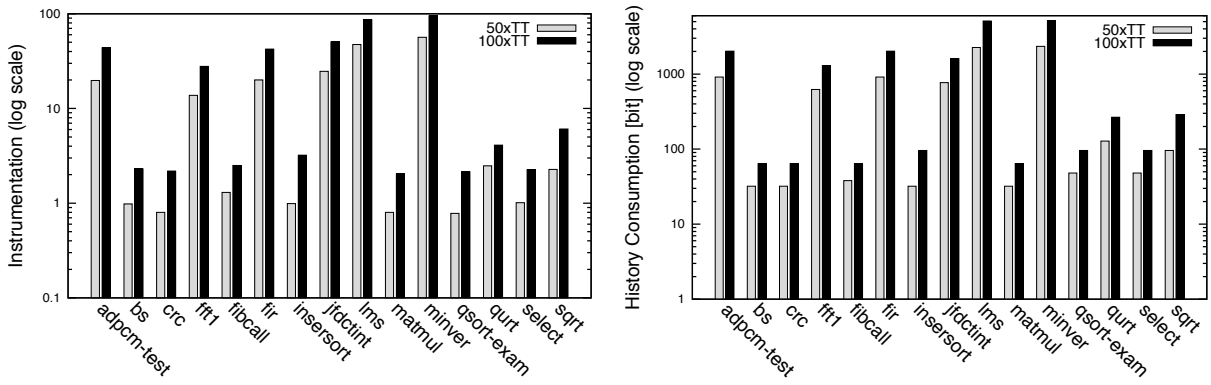


(b) Redundant samples of TT monitor



(c) Frequency of redundant samples

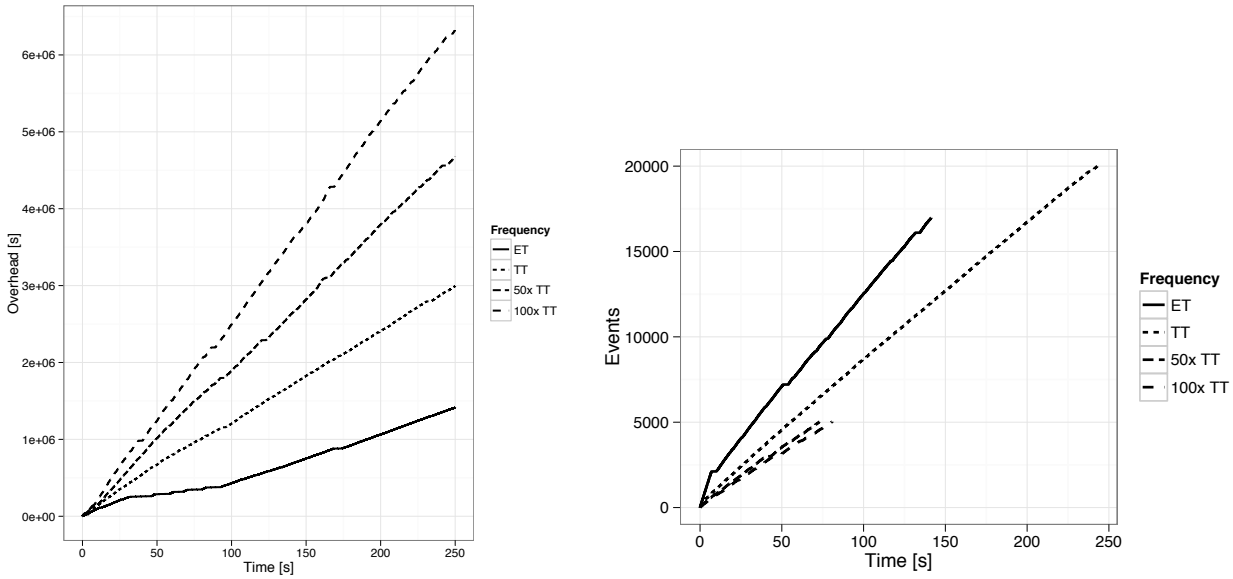
Figure 5.6: Redundant samples and their frequency



(a) Number of variables stored in the history

(b) Amount of history consumption

Figure 5.7: Number of variables stored in history and memory consumption



(a) Cumulative overhead measured for sqrt

(b) Bound on monitoring events for sqrt

Figure 5.8: Resource management

Algorithm 1 Greedy

Input: A critical control-flow graph $G = \langle V, v^0, A, w \rangle$ and desired sampling period SP.
Output: A set U of vertices to be deleted from G.

```
1: U := {};
2: G := PruneCFG(G, SP);

3: while (MW(G) < SP ∧ U ≠ V) do
4:     v := GreedySearch(G);
5:     G := CollapseVertex(G, v);
6:     U := U ∪ {v};
7: end while

8: if (U = V) then declare failure;
9: return U;
```

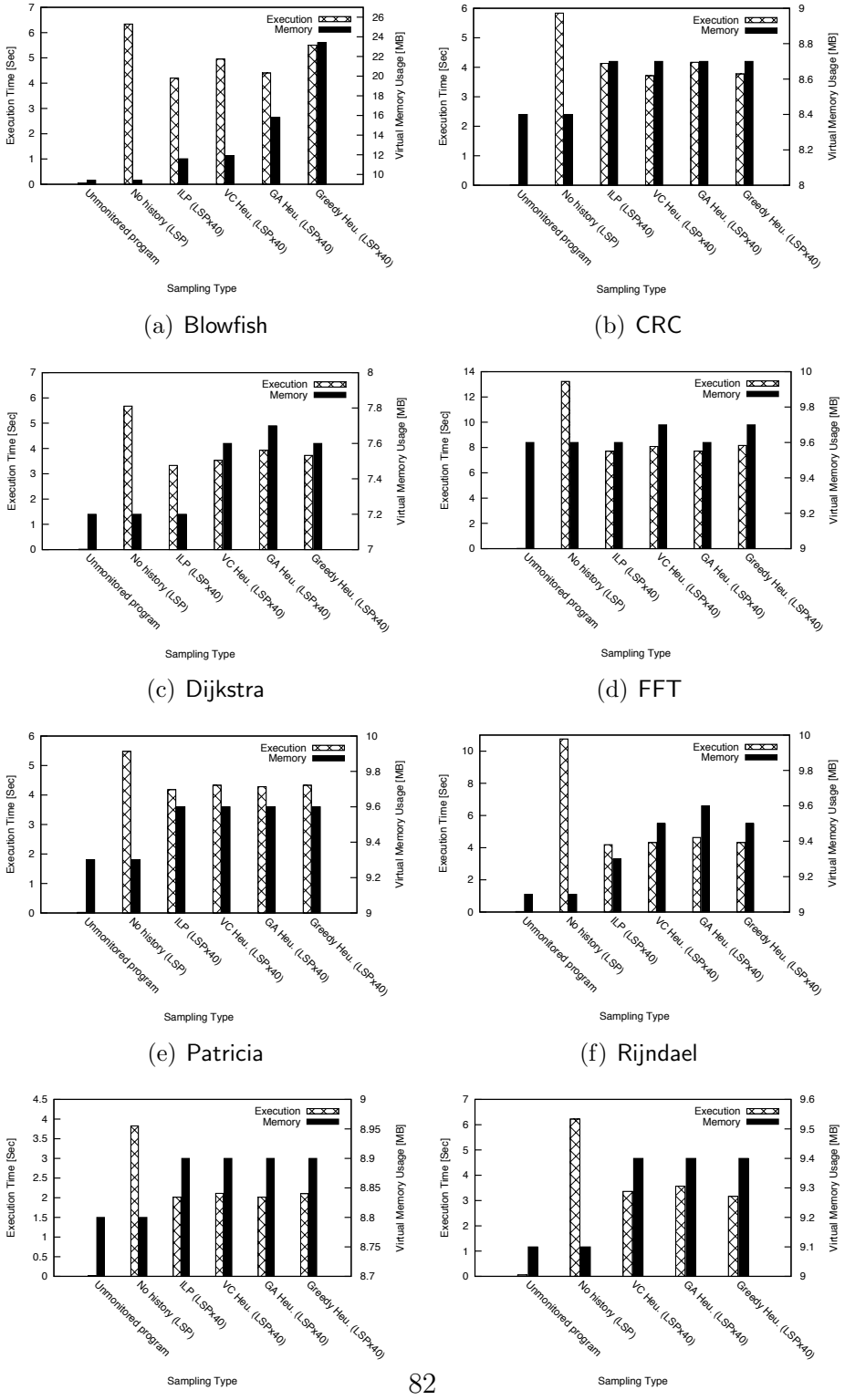
Algorithm 2 Vertex Cover Based

Input: A critical control-flow graph $G = \langle V, v^0, A, w \rangle$ and desired sampling period SP.
Output: A set U of vertices to be deleted from G.

```
1: U := {};
2: G := PruneCFG(G, SP);

3: while (MW(G) < SP ∧ U ≠ V) do
4:     vc := Approximate-Vertex-Cover(G);
5:     for each vertex v ∈ vc do
6:         G := CollapseNode(G, v);
7:         U := U ∪ {v};
8:     end for
9: end while

10: if (U = V) then declare failure;
11: return U;
```



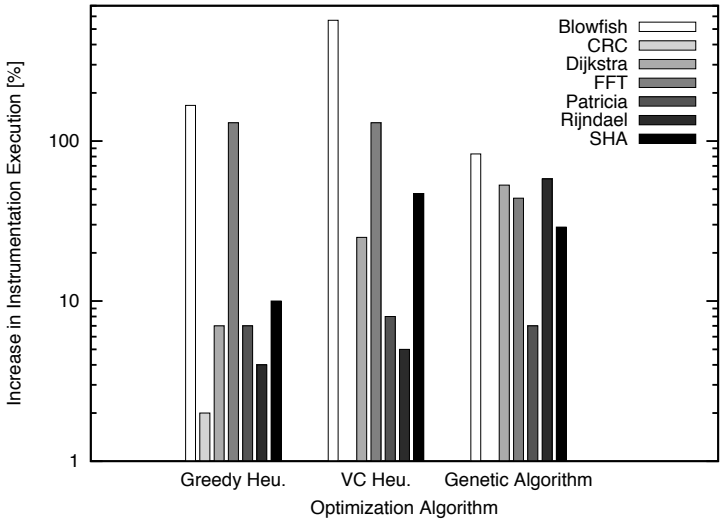
(a) Blowfish (b) CRC

(c) Dijkstra (d) FFT

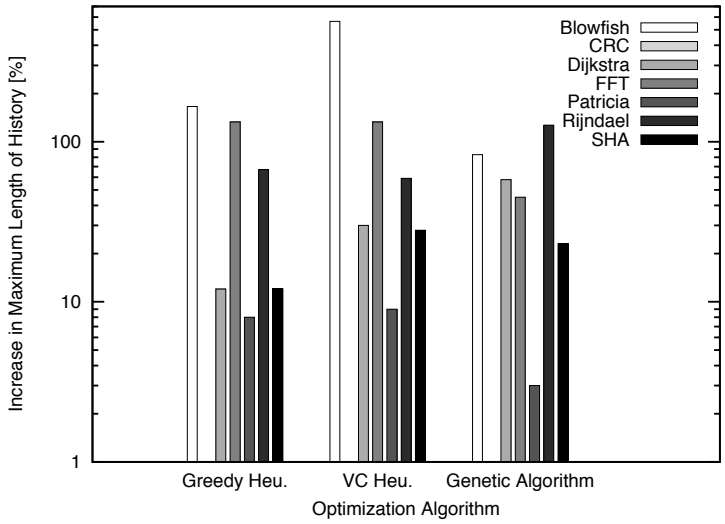
(e) Patricia (f) Rijndael

(g) Sha (h) Susan

Figure 5.9: The impact of sampling types on memory and execution time



(a) Increase in the number of execution of instrumentation instructions



(b) Increase in the maximum size of history between two samples

Figure 5.10: The impact of sub-optimal solutions on execution of instructions to build history and its maximum size

Chapter 6

Path-aware Time-triggered Monitoring

In Chapter 5, our experimental results show that our time-triggered monitor without history successfully satisfies the following three characteristics that are required for runtime verification of real-time embedded systems (see Chapter 3).

1. Predictable monitoring,
2. Bounded monitoring overhead, and
3. Reduced over-provisioning.

On the contrary, our time-triggered monitor without history does not provide *efficient monitoring overhead*. Chapter 5 shows that the main cause of the excessive overhead is *redundant monitoring samples*. Recall that a redundant sample is when the sampler takes a sample from the program when no critical instruction has been executed since the sampler's last sampling point. To this end, in Chapter 5, we present the use of history (i.e., auxiliary memory) to reduce the number of redundant samples by increasing the longest sampling period of the program under scrutiny. Experimental results from Chapter 5 show that by using history, we can increase the longest sampling period and hence, reduce the redundant samples by up to 90%. To this end, our time-triggered monitor with history provides efficient monitoring overhead. Despite the experimental results , using history does not eliminate the main source of redundant samples.

Our studies show that the main source of redundant samples is the technique we use to calculate the longest sampling period (see Section 4.2.1). Recall that we use the *complete* critical control-flow graph to calculate the longest sampling period, disregarding the execution path and hence, the sequence of critical instructions that the program under scrutiny takes at run time.Thus, the calculated longest sampling period tends to be conservative which results in the time-triggered monitor taking redundant samples. From this point on, we will refer to the longest sampling period calculated by the technique in Section 4.2.1 as the *fixed LSP*.

To clarify, consider the Fibonacci program in Figure 6.1(a) and its control-flow graph and critical control-flow graph in Figures 6.1(b) and 6.1(c) respectively. The fixed longest sampling period of Fibonacci with respect to $\mathcal{V}_\Pi = \{\mathsf{Fnew}, \mathsf{Fold}, \mathsf{ans}\}$ is equal to 1 time unit. This sampling period is *optimal* when Fibonacci is given the input $\mathsf{n} = 2$ and executes the instruction sequence $\langle v_A, v_D, v_{C_1}, v_{C_2}, v_{C_3}, v_{C_4}, v_D, v_{B_1}, v_{B_2}\rangle$ at run time. On the contrary, when the program is given the input $\mathsf{n} = 0$ and hence, executes the instruction sequence $\langle v_A, v_D, v_{B_1}, v_{B_2}\rangle$, the fixed longest sampling period is conservative. In the latter case, the sampler can extract all values of variables of interest with a longest sampling period of 5 time units. In this case, the sampler takes 85% less redundant samples compared to using the fixed longest sampling period of 1 time unit.

The aforementioned results motivate the idea of computing the longest sampling period of the program under scrutiny with respect to its execution path at run time. To this end, we aim at solving the following problem:

Problem statement. Given a set of properties Π (e.g. LTL formulas) and an execution path $E = \langle v_0, v_1, v_2, ...\rangle$ of a program P, find the longest sampling period (LSP) which enables the monitor to provide sound runtime verification with respect to E (i.e., all of the critical instructions of execution path E are sampled).

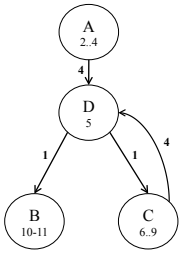
In this Chapter, we present two approaches:

1. *Path-aware approach* [82]: This approach predicts the execution path of the program with respect to its input. Consequently, it uses the approach from Section 4.2.1 to calculate the execution path's longest sampling period.

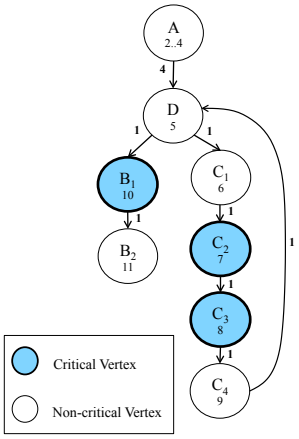
2. *Adaptive Path-aware approach* [82]: This approach adjust the longest sampling period of the time-triggered monitor at run time based on the sequence of critical instruction *soon-to-be* executed within the program's predicted execution path.

```
1. fib(int n) {
2.     int  i, Fnew, Fold, temp,ans;
3.     Fnew = 1;  Fold = 0;
4.     i = 2;
5.     while( i <= n ) {
6.         temp = Fnew;
7.*        Fnew = Fnew + Fold;
8.*        Fold = temp;
9.         i++; }
10.*   ans = Fnew;
11.    return ans;}
```

(a) Fibonacci Function



(b) Control-flow graph



(c) Critical control-flow graph

Figure 6.1: Fibonacci and its control-flow graph and critical control-flow graph

6.1 Path-aware Longest Sampling Period

The Fibonacci example motivates the idea to compute the longest sampling period with respect to the execution path that the program takes at run time. To this end, we must carry out the following two steps to be able to calculate the longest sampling period of an execution path:

1. Predict the execution path that the program will take at run time with respect to the program's input values.

2. Using the predicted path from Step 1, compute the longest sampling period by only considering the sequence of critical instructions within the execution path.

6.1.1 Path Prediction

To be able to statically calculate the longest sampling of an execution path of a program, we must predict the execution path the program will take at run time with respect to the set of inputs which will be given to the program. To this end, we formalize the *path prediction* problem as follows.

Let P be a program, $CFG_P = \langle V, v^0, A, w\rangle$ be its control-flow graph, and $\mathcal{I}_P$ be the *input domain* of P. The input domain is the set of all values that the environment (e.g., a user) can provide as input to P.

Definition 9 (Execution Path) *An* execution path *is a sequence of the form* $\gamma = \langle(v_0, \omega_0, v_1), (v_1, \omega_1, v_2), ...\rangle$, *where:*

- $v_0 = v^0$.
- *For all* $i \geq 0$, $v_i \in V$.
- *For all* (v_i, ω_i, v_{i+1}), *where* $i \geq 0$, *there exists an arc* (v_i, v_{i+1}) *in* A.
- *For all* $i \geq 0$, $\omega_i = w(v_i)$.
- *If* P *is a terminating program, then* $\gamma = \langle(v_0, \omega_0, v_1), \ldots, (v_{n-1}, \omega_{n-1}, v_n)\rangle$ *is finite and* v_n *is a vertex in* V *with outdegree of zero.* □

For instance, in Fibonacci, the input value n=0 leads to the execution of path $\gamma_1 = \langle(v_A, 4, v_D), (v_D, 1, v_{B_1}), (v_{B_1}, 1, v_{B_2})\rangle$. Note that in this thesis, we only focus on *possible* execution paths. A possible execution path is an execution path for which there exists some input in $\mathcal{I}_P$ that enables P to take the execution path at run time. We denote the set of all *possible* execution paths of program P as $\mathcal{P}_P$.

To *predict* the execution path(s) of P with respect to a set of inputs from its input domain $\mathcal{I}_P$, we require a mechanism that takes the value-set of the inputs of P and returns the set of execution paths of P. We refer to this mechanism as the *path prediction* function.

Definition 10 (Path Prediction Function) *Let P be a program. The* path prediction *function $\psi_P : \mathcal{I}_P \rightarrow 2^{\mathcal{P}_P}$, maps an input from the input domain of P to a subset of execution paths of P.* □

Note that in a deterministic program, ψ_P maps an input to one and only one execution path. In practice, ψ_P can be implemented using symbolic execution [60] before the actual run of the program. In particular, symbolic execution creates a bijection from each execution path γ of a program to a *path constraint*. A path constraint projects the conditions (e.g., in if-then-else and loop structures) that need to be satisfied in order for the program to execute γ at run time.

Notation: For each path γ, $PC(\gamma)$ denotes the path constraint of γ. For instance, for execution path $\gamma_1 = \langle (v_A, 4, v_D), (v_D, 1, v_{B_1}), (v_{B_1}, 1, v_{B_2}) \rangle$ of the Fibonacci program, the path constraint of γ_1 is as follows, $PC(\gamma_1) = (n < 2)$. Thus, ψ_P in fact, uses the input values (e.g., n = 0) and $PC(\gamma)$ to find the path constraint(s) satisfied by the input values, and extracts the set of associated execution path(s).

6.1.2 Computing the Longest Sampling Period

Given a predicted execution path γ from Section 6.1.1, the longest sampling period of γ is computed as follows. We refer to the following longest sampling period as *path-aware longest sampling period (paLSP)*.

Definition 11 (path-aware longest sampling period) *Let $x \in \mathcal{I}_P$ be an input and the set of possible predicted execution paths be $\psi_P(x) = \{\gamma\}$. The* path-aware longest sampling period *paLSP for γ is the minimum length subpath between any two critical vertices of γ.* □

Now, consider a program that includes a loop structure. It is likely that an execution path γ of the program has multiple occurrences of a subpath of the form $\langle (v_i, \omega_i, v_{i+1}), (v_{i+1}, \omega_{i+1}, v_{i+2}), ..., (v_n, \omega_n, v_i) \rangle$. We refer to such a subpath as a *loop sequence*. Observe that multiple occurrences of a loop sequence in γ do not affect the value of *paLSP*. Hence, before computing *paLSP*, our approach transforms γ such that each of its loop sequences consecutively occur at most *twice*. We refer to the resulting execution path as the *unique* version of γ.

Definition 12 (Unique Execution Path) *Let γ be an execution path. The* unique execution path *of γ, denoted γ^{unq}, is a path, where each consecutive occurrence of at least 2 for a loop sequence L in γ is represented by 2 consecutive occurrences of L in γ^{unq}.* □

We refer to a time-triggered runtime verification framework whose sampler uses path-aware longest sampling period *paLSP*, as *path-aware time-triggered runtime verification framework* (pa-TTRV).

Algorithm 3 calculates the *paLSP* of an execution path. This algorithm takes the program's control-flow graph, the unique execution path γ^{unq} and set of variables of interest as input. At Line 1, it uses the SUBCFG function to extract the portion (sub-control-flow graph) of the program's control-flow graph that is executed by γ^{unq}. At Line 2, the algorithm calls function CCFG that uses the technique from Section 4.2.1 to calculate the critical control-flow graph of the sub-control-flow graph $CFG_{\gamma^{unq}}$. Then, at Lines 4-10, the algorithm iterates over all possible pairs of consecutive critical vertices v_i and v_j in $CFG_{\gamma^{unq}}$ to extract the minimum subpath between all possible pairs. At Line 5, the algorithm uses function SUBPATH which extracts all the subpaths between critical vertices v_i and v_j. Note, that if there are multiple subpaths between v_i and v_j, SUBPATH only returns the path with the least weight. At Line 6, it extracts the weight of the subpath $SubP$. In the end, at Lines 7-9, Algorithm 3 stores the weight of the subpath with the minimum weight and returns it at the end.

Algorithm 3 Calculating *paLSP*

Input: CFG: control-flow graph of program P, γ^{unq}: a unique execution path, $\mathcal{V}$: variables of interest

Output: *paLSP* of γ

1: $CFG_{\gamma^{unq}} \leftarrow \mathsf{SUBCFG}(CFG, \gamma^{unq})$ /* *extract CFG covered by* γ */
2: $CCFG_{\gamma^{unq}} \leftarrow \mathsf{CCFG}(CFG_{\gamma^{unq}}, \mathcal{V})$ /* *extract critical control-flow graph* */
3: $minW \leftarrow \infty$
/* *iterate over the critical vertices of* $CCFG_{\gamma^{unq}}$ */
4: **for** any two critical vertices v_i and v_j of $CCFG_{\gamma^{unq}}$ **do**
5: $SubP \leftarrow \mathsf{SUBPATH}(v_i, v_j)$ /* *extract subpath between* v_i *and* v_j */
6: $w \leftarrow \sum_{v \in V_{SubP}} w(v)$ /* *extract weight of subpath* */
/* *Find minimum subpath* */
7: **if** $w < minW$ **then**
8: $minW \leftarrow w$
9: **end if**
10: **end for**
11: **return** $minW$

6.2 Adaptive Path-aware Longest Sampling Period

Although our experiments (see Section 6.4) show that pa-TTRV can effectively reduce the number of redundant samples, it still imposes excessive redundant samples when only a small fraction of the execution path needs to be sampled with the computed *paLSP*. For instance, if Fibonacci (see Figure 6.1) takes the hypothetical execution path $\gamma_2 = \langle v_A, v_D, v_{B_1}, v_{B_2}, v_A, v_D, v_{C_1}, v_{C_2}, v_{C_3}, v_{C_4}, v_D, v_{B_1}, v_{B_2} \rangle$, the path-aware longest sampling period *paLSP* is 1 time unit. However, if we apply a sampling period of 5 time units (i.e., $paLSP = 5$) up until vertex v_{C_2} (where all critical instructions can be sampled) and adjust the sampling period to 1 time unit (i.e., $paLSP = 1$) afterwards, then the number of samples drops by 62%. With this motivation, we present a technique that *regionalizes* the predicted execution path into *regions* with respect to the distribution of critical instructions throughout the execution path.

6.2.1 *LSP* Regions

Intuitively, our idea to reduce redundant samples in an execution path is to dynamically change the path-aware longest sampling period *paLSP* according to the *LSP regions* of the execution path.

Definition 13 (LSP Region) *Let γ be a unique execution path. An* LSP region *is a set of subpaths of γ with the same path-aware longest sampling period (paLSP), where each subpath is* maximal. *That is, for each LSP region, if a subpath is extended, it no longer belongs to that region.* □

Since each subpath has a unique path-aware longest sampling period *paLSP*, each *LSP* region is an equivalence class. It is straightforward to observe that a binary *paLSP* equality relation defined over a code region is reflexive, symmetric, and transitive.

To sample an execution path with an *adaptive path-aware longest sampling period* (*adaptive paLSP*), our approach needs to somehow *regionalize* the path based on Definition 13. In this case, when the program starts executing a subpath of an *LSP* region, the monitor *adapts* to the longest sampling period *paLSP* of that *LSP* region at run time. Clearly, the regionalization can partition an execution path in various ways. Our general objective is to regionalize an execution path such that adapting the longest sampling period at run time does not add excessive overhead. We break down our objective as follows:

1. Reducing the number of *LSP* regions; i.e., since change of *LSP* region and hence, sampling period at run time incurs some overhead. In other words, the cost of switching the sampling period should not impose significant overhead.

2. Reducing the number of samples taken on the execution path and hence, reducing the number of redundant samples.

3. Maintaining the absolute jitter of *paLSP* (i.e., the difference between the minimum and maximum *paLSP* of *LSP* regions) below a predefined threshold Δ_{LSP} provided by the designer. Note that this objective ensures *predictable monitor invocation*. Recall that one of the goals for using a time-triggered monitor is to achieve predictable monitoring (see Chapter 3).

We refer to a time-triggered runtime verification framework whose sampler uses adaptive path-aware longest sampling period (i.e., adaptive *paLSP*), as *adaptive pa-TTRV*.

6.2.2 A Regionalization Algorithm

The algorithm to create a partition that satisfies the above objectives has a complexity of $O(n^2)$. The algorithm Regionalize addresses the above objectives (see Algorithm 4). It takes as input:

1. The bound Δ_{LSP} on the absolute jitter,

2. The overhead of changing the sampling period O_{LSP}, and

3. A unique execution path γ.

Its output is a regionalization. The intuitive idea is that the algorithm creates all possible regionalizations and chooses the one with the least monitoring overhead.

The algorithm creates all possible regionalizations using three nested loops. Each iteration of each loop creates a new regionalization, where each regionalization is different from the other (created in the same loop) by one vertex.

1. The for-loop (Lines 4-38). The algorithm partitions γ into three subpaths at each iteration of the loop at line 4. This loop puts the first subpath $\langle(v_0, \omega_0, v_1), ..., (v_i, \omega_i, v_{i+1})\rangle$ into a separate code region (line 6).

2. The while-loop (Lines 8-37). This loop takes the second subpath and puts each of the entities (i.e., (v, ω, v')) into separate code regions (lines 10- 13).

3. The for-loop (Lines 15-35). This loop further partitions the third subpath into two parts. Initially, the loop puts the prefix $\langle(v_{base}, \omega_{base}, v_{base+1}), ..., (v_m, \omega_m, v_{m+1})\rangle$ of the third subpath into one code region (lines 14- 19). As for the suffix $\langle(v_{m+1}, \omega_{m+1}, v_{m+2}), ..., (v_{n-1}, \omega_n, v_{n-1})\rangle$, the loop puts each remaining entities in separate code regions (lines 20- 23).

Notice that the first loop (i.e., first for-loop) adds/removes vertices from subpath $\langle(v_0, w_0, v_1), \ldots, (v_i, w_i, v_{i+1})\rangle$, the second loop (i.e., the while-loop) adds/removes vertices from subpath $\langle(v_{i+1}, w_{i+1}, v_{i+2}), \ldots, (v_{base-1}, w_{base-1}, v_{base})\rangle$, and the third loop (i.e., the second for-loop) adds/removes vertices from subpath $\langle(v_{base}, w_{base}, v_{base+1}), \ldots, (v_{n-1}, w_{n-1}, v_n)\rangle$.

When a regionalization $temp_{reg}$ is created (Line 23), the algorithm computes the monitoring overhead of $temp_{reg}$. To this end, it computes *paLSP* (line 26), and the best case execution time of each *LSP* region of the regionalization (line 27). Respectively, it computes the monitoring overhead by considering the number of samples taken in each *LSP* region and the cost of changing *LSP* regions (line 28). If $temp_{reg}$ has a lower monitoring overhead compared to the previously chosen regionalization, and the absolute jitter of *paLSP*s is bounded by Δ_{LSP}, the algorithm chooses $temp_{reg}$ as the solution (lines 31- 34).

Algorithm 4 Regionalize

Input: Δ_{LSP}:bound on absolute jitter, O_{LSP}: overhead of changing LSP regions, γ: a unique execution path

Output: A regionalization

1: $regionalization \leftarrow \emptyset$
2: $Overhead_{reg} \leftarrow \infty$
3: $n \leftarrow Length(\gamma)$
   // *iterate over the vertices of* γ
4: **for** $i = 0$ to $n - 1$ **do**
5: $temp_{reg} \leftarrow \emptyset$ /* *new regionalization* */
6: $reg \leftarrow \langle (v_0, \gamma_0, v_1), ..., (v_i, \gamma_i, v_{i+1}) \rangle$
   // *regionalization of remainder of* γ
7: $base \leftarrow i + 1$
8: **while** $(base \leq n - 1)$ **do**
9: $temp_{reg} \leftarrow temp_{reg} \cup reg$
   // *put vertex in* γ *up to base in separate regions*
10: **for** $m = i + 1$ to $base - 1$ **do**
11: $reg' \leftarrow \langle (v_m, \gamma_m, v_{m+1}) \rangle$
12: $temp_{reg} \leftarrow temp_{reg} \cup reg'$
13: **end for**
14: $reg'' \leftarrow \langle (v_{base}, \gamma_{base}, v_{base+1}) \rangle$
15: **for** $m = base$ to $n - 1$ **do**
16: **for** $j = base + 1$ to m **do**
17: $reg'' \leftarrow append(reg'', (v_j, \gamma_j, v_{j+1}))$ /* iterates only when $m > base$ */
18: **end for**
19: $temp_{reg} \leftarrow temp_{reg} \cup reg''$
   // *put the remainder of* γ *in separate regions*
20: **for** $q = m + 1$ to $n - 1$ **do**
21: $reg''' \leftarrow \langle (v_q, \gamma_q, v_{q+1}) \rangle$
22: $temp_{reg} \leftarrow temp_{reg} \cup reg'''$
23: **end for**
   // *calculate overhead for new regionalization*
24: $Overhead \leftarrow 0$
25: **for all** $reg \in temp_{reg}$ **do**
26: Compute $paLSP_{reg}$ based on Definition 11
27: $total \leftarrow$ The sum of weights of arcs in reg
28: $Overhead \leftarrow Overhead + \frac{paLSP_{reg}}{total} + O_{LSP}$
29: **end for**
30: $\Delta_{reg} \leftarrow$ Absolute jitter of $paLSP$s of $temp_{reg}$
   // *Update the best regionalization*
31: **if** $Overhead < Overhead_{reg}$ **and** $\Delta_{reg} \leq \Delta_{LSP}$ **then**
32: $regionalization \leftarrow temp_{reg}$
33: $Overhead_{reg} \leftarrow Overhead$
34: **end if**
35: **end for**
36: $base \leftarrow base + 1$
37: **end while**
38: **end for**
39: **return** *regionalization*

6.2.3 General Code Regionalization

Observe that Definition 13 and Algorithm 4 identify a regionalization for an execution path. Hence, if two execution paths in a program share a common subpath, the subpath does not necessarily reside in the same *LSP* region. For instance, consider the following execution paths: $\gamma_1 = \langle (v_0, 5, v_1), (v_1, 10, v_2), (v_2, 15, v_3) \rangle$ and $\gamma_2 = \langle (v_0, 5, v_1), (v_1, 1, v_5), (v_5, 2, v_6) \rangle$, where $\Delta_{LSP} = O_{LSP} = 5$. Algorithm 4 computes the following two *LSP* regions for γ_1:

1. $reg_1 = \{\langle (v_0, 5, v_1) \rangle\}$, where $paLSP_{reg_1} = 5$, and
2. $reg_2 = \{\langle (v_1, 10, v_2), (v_2, 15, v_3) \rangle\}$, where $paLSP_{reg_2} = 10$.

On the contrary, for γ_2, Algorithm 4 computes a single *LSP* region $reg_{\gamma_2} = \{\gamma_2\}$, where $paLSP_{reg_{\gamma_2}} = 1$. Hence, subpath $\langle (v_0, 5, v_1) \rangle$ resides in different regions with different *paLSP*s for execution paths γ_1 and γ_2. Thus, in environments where a unique regionalization among all execution paths of the program is desirable, we generalize the regionalization process as follows.

Definition 14 (General Regionalization) *Let* $CFG = \langle V, v^0, A, w \rangle$ *be a control-flow graph. In* general regionalization*, each arc* $(u, v) \in A$ *appears in one and only one LSP region.* □

In this case, the monitor *adapts* the *paLSP* of an *LSP* region *reg* at run time when:

1. The program initiates the execution of a subpath in *reg*, and
2. *reg* differs from the *LSP* region of the previously executed subpath.

Obtaining a general regionalization that optimally satisfies the three objectives mentioned in Subsection 6.2.1 has exponential complexity. In Section 6.3, we present a heuristic that provides an efficient approach to implement general regionalization.

6.3 Implementation

In this section, we present the tool chain that computes *paLSP* and adaptive *paLSP*, along with the implementation of the time-triggered monitor.

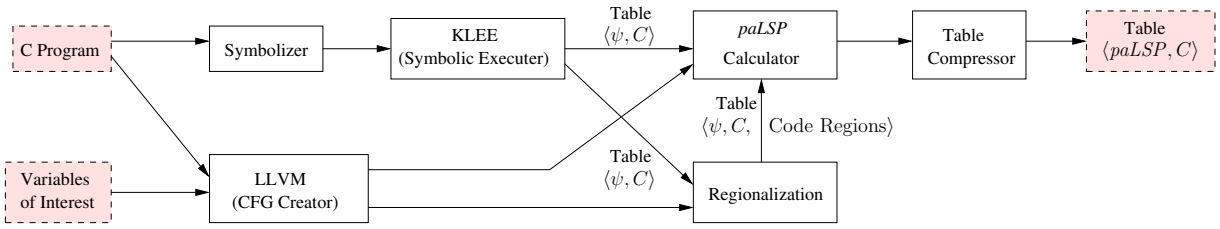


Figure 6.2: Tool chain for calculating *paLSP* and adaptive *paLSP*

6.3.1 Tool Chain

We have implemented a tool chain for computing *paLSP* and adaptive *paLSP* (see Figure 6.2). The tool's input is a C program and a set of variables of interest. First, it extracts the control-flow graph and subsequently the critical control-flow graph of the program using the *CFG creator* module. This module is implemented over LLVM and takes advantage of LLVM's built in transformation passes. This module is in fact the *CFG* phase from the tool chain in Subsection 5.3.2 (see Figure 5.2).

In order to implement the path prediction function ψ_P (see Definition 10), the tool chain first symbolizes the input variables of the program using the *Symbolizer* module. In other words, *Symbolizer* converts the input variables to symbols that can be interpreted by the symbolic execution tool. With this respect, all the path constraints will be based on the value of the input variables. Then, it feeds the symbolized program to the symbolic execution tool KLEE [24]. KLEE runs the program and extracts the path constraint of each possible execution path of the program with respect to the symbolized variables. The path constraints are STP [3] regular expressions over the symbolized variables. In the end, KLEE creates a mapping table from each unique execution path of the program to its path constraint. We refer to this table as the *path table*. Note that we modified KLEE using a patch, such that it converts an execution path to its unique version (see Definition 12) before adding it to the path table. In case there exist duplicate unique paths, our KLEE patch only keeps the path with the *weakest* path constraint.

To compute the adaptive *paLSP*, the tool chain regionalizes the program using general regionalization (see Definition 14). Recall that computing the optimal general regionalization has exponential complexity. To this end, in this set of experiments, the *Regionalization* module uses a simple greedy heuristic that considers all the arcs in between two consecutive conditional statements (e.g., If-then-else, for, while statements) in the control-flow graph of the program as one *LSP* region. For instance, the control-flow graph of the Fibonacci code (see Figure 6.1(b)) has three *LSP* regions: (1) $reg_1 = \{\langle(v_A, 4, v_D)\rangle\}$,

(2) $reg_2 = \{\langle(v_D, 1, v_C), (v_C, 4, v_D)\rangle\}$, and (3) $reg_3 = \{\langle(v_D, 1, v_B)\rangle\}$. Consequently, the Regionalization module maps each unique execution path in the path table to the set of its *LSP* regions. The *paLSP* Calculator module uses the critical control-flow graph to compute *paLSP* and adaptive *paLSP* of each unique execution path in the path table created by KLEE. For adaptive *paLSP*, the *paLSP* calculator computes the *paLSP* of each *LSP* region in each execution path.

In general, the size of the path table (i.e., execution path to path constraint mapping table) of KLEE may grow exponentially with respect to the number of execution paths. This implies that looking up the path table at run time imposes a large overhead. Thus, it is desirable to construct a smaller version of the path table to be used at run time. To this end, the tool chain applies two techniques (i.e., *Table Compressor* module) to eliminate entries:

1. *Implication Reduction*: This technique groups the execution paths whose *paLSP* is defined by the same arc (u, v) in the critical control-flow graph. For each group, it extracts the path constraint whose satisfaction leads to the execution of (u, v). Then, it represents the execution paths in the group with a single table entry that maps the extracted path constraint to *paLSP* of the execution paths in the group. This table entry also incorporates the *union* of the set of *LSP* regions of the execution paths in the group. To clarify, consider the following three hypothetical execution paths of a program P:

   - $\gamma_1 = \langle(v_A, 10, v_{B_1}), (v_{B_1}, 10, v_{E_1}), (v_{E_1}, 1, v_{E_2}), (v_{E_2}, 10, v_{B_2}), (v_{B_2}, 10, v_Z)\rangle$
   - $\gamma_2 = \langle(v_A, 20, v_{C_1}), (v_{C_1}, 20, v_{E_1}), (v_{E_1}, 1, v_{E_2}), (v_{E_2}, 20, v_{C_2}), (v_{C_2}, 20, v_Z)\rangle$
   - $\gamma_3 = \langle(v_A, 5, v_{D_1}), (v_{D_1}, 5, v_{E_1}), (v_{E_1}, 1, v_{E_2}), (v_{E_2}, 5, v_{D_2}), (v_{D_2}, 5, v_Z)\rangle$

   As can be seen, all three execution paths have $paLSP = 1$ that is defined by the arc $(v_{E_1}, 1, v_{E_2})$. To this end, the implication reduction technique represents paths γ_1, γ_2, and γ_3 via the following single entry in the path table: $\{paLSP = 1,\ PC = PC(\gamma_1) \vee PC(\gamma_2) \vee PC(\gamma_2),\ LSP \text{ regions} = \bigcup_{\gamma=\gamma_1}^{\gamma_3} LSP\_Regions_\gamma\}$.

2. *paLSP Reduction*: This technique removes all entries from the mapping table, where *paLSP* of the execution path is similar to the fixed *LSP* when only using the *paLSP* approach. For instance, if the fixed *LSP* of program P is 1 time unit, then *paLSP* Reduction will remove γ_1, γ_2, and γ_3 from the path table. In this case, when the input values do not satisfy any of the path constraints in the path table, the sampling period of the sampler is set to the fixed *LSP*. As can be seen, *paLSP* Reduction does not change the behavior of *paLSP* approach.

The final path table maps a path constraint to a *paLSP* and a set of *LSP* regions along with their *paLSP*. These techniques, on average, reduce the mapping table of SNU programs by 78% without loss of precision. In other words, the *paLSP* and the adaptive *paLSP* of an execution path do not change.

In cases where KLEE cannot process all execution paths because of its limitations, the tool chain takes two conservative approaches.

1. When there is unanalyzed code, it assumes that this code is executed at all times and hence, appends it to all execution paths.

2. When the analysis of an execution path γ is incomplete, the tool finds all possible unique subpaths that can be executed after γ and hence, creates new paths by appending these subpaths to γ.

6.3.2 Implementing a Path-aware Time-triggered Sampler

The time-triggered sampler is a C program which runs in parallel with the program under scrutiny. The sampler has read access to the memory location of the variables of interest. The sampler has three modes: *fixed*, *path-aware*, and *adaptive*. In the fixed mode, the sampler samples the program using the fixed *LSP* which is calculated via the technique from Section 4.2.1 . In the path-aware mode, at each point at run time, where the program receives a new input from the environment[1], the sampler looks up the path table created by KLEE and evaluates path constraints to identify the proper path-aware longest sampling period *paLSP*. Then, the sampler adjusts the sampling period of the time-triggered sampler accordingly. In the adaptive mode, the time-triggered sampler applies all features of the path-aware mode. In addition, the program is instrumented, so that when it reaches a new *LSP* region, it notifies the sampler to adjust its sampling period accordingly with respect to the path table which maps each *LSP* region to its *paLSP*.

6.4 Experimental Results

In this section, we present the experimental results from a time-triggered monitor whose sampler uses *paLSP* or adaptive *paLSP* as its sampling period. We use programs from the SNU benchmark [2] to evaluate our approaches.

[1]Examples include executing instructions such as scanf, read, fscanf, etc.

6.4.1 Experimental Settings

In each program, the main function runs 100 times, where at each iteration the main function receives new input values from the environment. The input values are such that each unique execution path of the program executes at least once. The program and the time-triggered sampler run on an MCB1700 board with RTX real-time operating system. The time-triggered sampler runs in four modes: (1) fixed *LSP*, (2) path-aware *LSP*, (3) adaptive *paLSP*, where Δ_{LSP} and O_{LSP} are 50ns, and (4) sampling periods of 50 $\times$ fixed *LSP*, 50 $\times$ *paLSP*, and 50 $\times$ adaptive *paLSP*, where the program is augmented with history and instrumented using our ILP approach (see Section 5.3) . We measure the following metrics to evaluate our approaches:

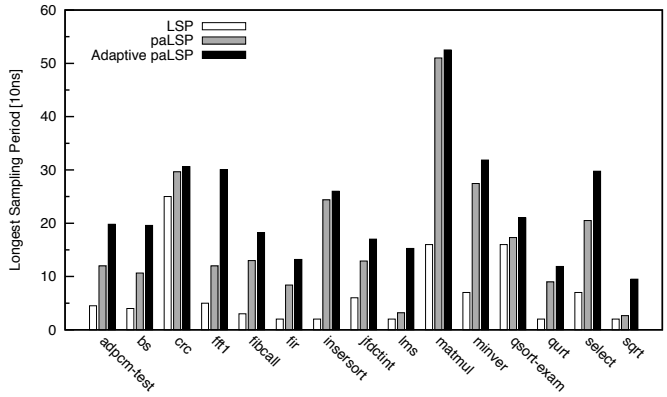
1. The values of the fixed *LSP*, *paLSP*, and adaptive *paLSP*.
2. The number of redundant samples taken at run time by the sampler.
3. The execution time of the monitored program. This value projects the amount of cumulative monitoring overhead imposed at run time. Recall that we are only interested in the overhead imposed by the sampler in this thesis.

6.4.2 Sampling Period of pa-TTRV and Adaptive pa-TTRV

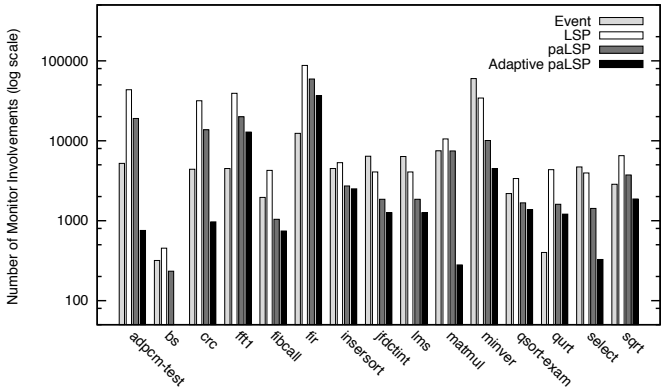
Figure 6.3(a) shows the fixed *LSP*, *paLSP*, and adaptive *paLSP* of each program. The *paLSP* of each program is the average path-aware longest sampling period *paLSP* over all unique execution paths of the program. As for the adaptive *paLSP* for each unique execution path, we consider the average path-aware longest sampling period *paLSP* over all the *LSP* regions of the execution path. Respectively, the adaptive *paLSP* of each program is the average adaptive *paLSP* over all unique execution paths of the program. The results show that the *paLSP* and adaptive *paLSP* of all programs are on average 2.4 and 3.34 times greater than their fixed *LSP*.

Observe that in some programs, *paLSP* is considerably greater than the fixed *LSP* (e.g., in insertsort this is 12.2 times). Our studies show that such programs have at least one of the following characteristics:

- The majority of the execution paths do not incorporate critical instructions and hence, do not require monitoring. For instance, 66.66% of the execution paths of insertsort do not require monitoring.



(a) Longest Sampling Period



(b) Redundant Samples of fixed *LSP*, *paLSP*, and adaptive *paLSP*

Figure 6.3: Sampling period and redundant samples

- In the majority of the execution paths, the critical instructions are sparsely distributed and hence, the required sampling period is greater than the fixed *LSP*. For instance, for 50% of the execution paths of select, *paLSP* is 130ns while the fixed *LSP* of select is 70ns.

On the contrary, in programs such as sqrt and lms, *paLSP* is moderately larger than the fixed *LSP*. Our studies show that such programs have at least one of the following characteristics:

- The majority of the execution paths execute the two consecutive critical instructions that define the fixed *LSP* of the program and hence, their *paLSP* is equal to the

fixed *LSP*. For instance, for 75% of sqrt's execution paths, *paLSP* is 20ns which is the same as sqrt's fixed *LSP*.

- In the majority of the execution paths, the critical instructions are densely distributed. For instance, 54% of the execution paths of lms have *paLSP* of 40ns while lms's fixed *LSP* is 20ns.

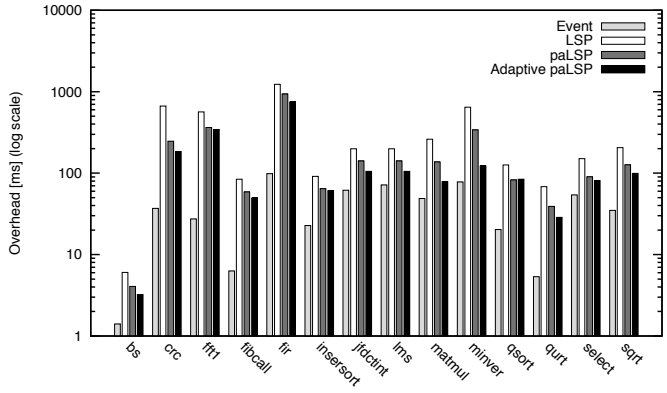
In addition, in programs such as select, adaptive *paLSP* is considerably greater than *paLSP*. In the execution path of such programs, the critical instructions are densely concentrated in a small fraction of the execution path, while in the remainder of the path, the critical instructions are sparse. For instance, in select, the critical instructions *only* reside in the function SWAP, where *paLSP* of an execution path executing SWAP is as low as 70ns. Hence, the adaptive *paLSP* of such execution paths is 48.75% larger than their *paLSP*.

6.4.3 Redundant Samples of pa-TTRV and Adaptive pa-TTRV

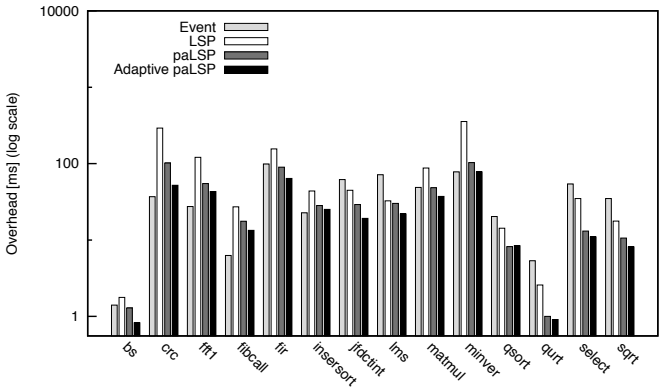
Figure 6.3(b) shows the number of redundant samples of a time-triggered monitor when using fixed *LSP*, *paLSP*, and adaptive *paLSP*. Note that the y-axis is in log scale. The *Event* bar shows the number of critical instructions executed throughout the program run. Bars *LSP*, *paLSP*, and adaptive *paLSP* show the number of redundant samples in fixed *LSP*, *paLSP*, and adaptive *paLSP* monitoring modes. The number of redundant samples is the difference between the total number of taken samples and the number of executed critical instructions. On average, by using *paLSP*, redundant samples decrease by 44.87%, and by using adaptive *paLSP*, redundant samples decrease by 64.04%.

Our analysis shows that in programs such as qurt and select, if the execution of paths with *paLSP* greater than the fixed *LSP* dominate the execution scenarios, then using *paLSP* results in larger reductions in the number of redundant samples. On the contrary, for programs such as sqrt and fir, we see small reduction in the number of redundant samples, since the majority of the executed paths at run time have *paLSP* equal to the fixed *LSP*. Note that a large percentage of paths with *paLSP* equal to the fixed *LSP* does not imply that the program's execution scenario is dominated by these paths.

To ensure that all critical instructions are sampled, we augment the program with a counter instCount. The program increments instCount after the execution of each critical instruction, and the sampler resets instCount at each sample. Our experiments show that the value of instCount is at most 1 at each monitoring sample which shows that at most one critical instruction had been executed since the last sample.



(a) Overhead of fixed *LSP*, *paLSP*, and Adaptive *paLSP*



(b) Overhead of 50 × *LSP*, 50 × *paLSP*, and 50 × adaptive*paLSP*

Figure 6.4: Monitoring overhead

6.4.4 Monitoring Overhead of pa-TTRV and Adaptive pa-TTRV

Figure 6.4(a) shows the monitoring overhead of a time-triggered monitor when using fixed *LSP*, *paLSP*, and adaptive *paLSP*. Each *Event* bar shows the execution time of the monitored program when using an event-triggered monitor. Bars *LSP*, *paLSP*, and adaptive *paLSP* show the execution time of the monitored program when monitored by a sampler with fixed *LSP*, *paLSP*, and adaptive *paLSP* monitoring modes. On average, monitoring overhead decreases by 39.34% when using *paLSP*, and by 51.28% when using adaptive *paLSP*.

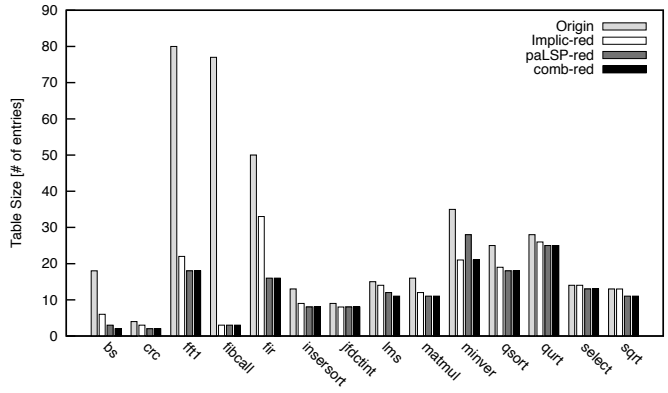
In programs such as qurt and select, when using *paLSP*, the monitoring overhead does

not decrease in the same proportion as the redundant samples. For instance, in select, the number of redundant samples decreases by 72.04%, while the monitoring overhead decreases 51.83%. This is because the overhead caused by the sampler to find the satisfied path constraint using the path table (see Section 6.3) and adjust its sampling period, is large. Hence, we see less reduction in the monitoring overhead.

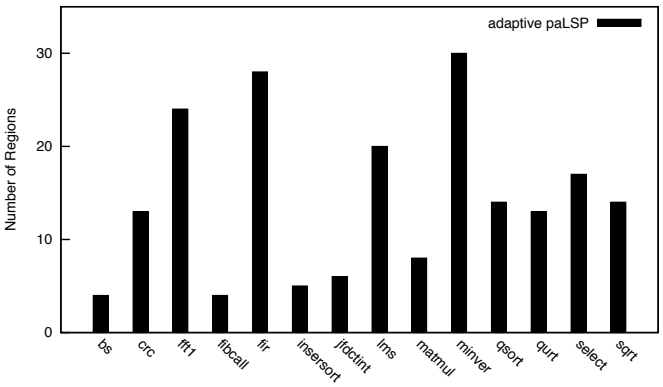
The same side effect is seen when using adaptive *paLSP* in programs such as select and fibcall. The overall overhead of looking up the path table in the adaptive path-aware sampler is larger compared to the path-aware sampler, since the sampler looks up the table more frequently (i.e., at each entry to a new *LSP* region). In some cases, the overall look up overhead is such that the monitoring overhead of the adaptive path-aware sampler exceeds the monitoring overhead of the path-aware sampler, although the adaptive path-aware sampler reduces more redundant samples. For instance, in qsort, the monitoring overhead of the adaptive path-aware sampler is 84.388ms, and the monitoring overhead of the path-aware sampler is 82.477ms, while the adaptive path-aware monitor removes 18.22% more redundant samples.

Figure 6.4(a) shows that event-based monitoring imposes less overhead than pa-TTRV and adaptive pa-TTRV when we do not use history to increase the program's fixed longest sampling period. This is because a time-triggered monitor still introduces redundant samples with *paLSP* and adaptive *paLSP*. Since event-based monitoring is impractical for real-time embedded systems, we need to further reduce the redundant samples to achieve a cost-worthy time-triggered monitor. To this end, we augment each program with history (see Section 4.2.1) to increase the fixed *LSP*, *paLSP*, and adaptive *paLSP* by a factor of 50. We have chosen the factor 50 because the experiments from Section 5.3.3 show that the factor 50 is sufficient to eliminate redundant samples in SNU programs and keep the behavior of the time-triggered monitor predictable. Experimental results show that for the SNU programs, the sampling periods of $50 \times$ fixed *LSP*, $50 \times$ *paLSP*, and $50 \times$ adaptive *paLSP* cause zero redundant samples, as expected.

Furthermore, Figure 6.4(b) shows that in 66% of the programs, the overhead of the path-aware monitor is less than the overhead of the event-triggered monitor, and in 75% of the programs, the overhead of the adaptive path-aware monitor is less than the overhead of the event-triggered monitor. Note that a sampling period of at least the *maximum* time interval between two consecutive critical instructions eliminates all redundant samples. In addition, by using history, the maximum increase in the memory usage of the programs is 646 bytes, which is an inconsiderable amount with respect to available resources in embedded systems.



(a) Size of lookup table when using *paLSP*



(b) Number of regions when using adaptive *paLSP*

Figure 6.5: Table size when using *paLSP* and adaptive *paLSP*

6.4.5 Size of Lookup Table

The experimental results from Subsection 6.4.4 shows that the size of the path table for *paLSP* and adaptive *paLSP* play a major role in the monitoring overhead. Figure 6.5(a) shows the size of the path table used by the path-aware sampler where each path constraint is mapped to a *paLSP*. In Figure 6.5(a), the *Origin* bar shows the number of entries that KLEE produces for the path table of each program without executing the patch which converts each path to a unique path. Recall from Section 6.3.1 that the Table Compressor module uses two distinct methods to reduce the size of the path table. The *Implic-red* bar shows the number of table entries after KLEE converts the paths into unique paths and the Table Compressor carries out the implication reduction phase. Results show that on

average, the implication reduction reduces the size of the path table by 27%. For instance, although in fft1 the implication reduction reduces the path table by 72%, the resulting table still has the considerable size of 22 entries. The *paLSP-red* bar shows the number of table entries after the Table Compressor carries out the *paLSP* reduction phase on the unique paths. Results show that on average, the *paLSP* reduction reduces the size of the path table by 34%. For instance, although in fir the implication reduction reduces the path table by 48%, the resulting table still has the considerable size of 26 entries. The *comb-red* bar shows the size of the path table after carrying out both the implication reduction and *paLSP* phase. Results show that on average, the combination of both techniques reduces the size of the path table by 38%. Note that, the overhead imposed by the path table not only depends on the size of the path table but it also depends on the overhead of evaluating the STP regular expression of each entry of the table.

Figure 6.5(b) shows the number of regions for each program when using general regionalization for carrying out adaptive *paLSP*. Recall that in this set of experiments, we consider the subgraph between two consecutive conditional statements (i.e., if-then-else, while, and for statements) as a single region. For small size programs, such as SNU programs, the number of regions are limited. For larger programs, we must devise a more intelligent heuristic compared to our current greedy approach to not only minimize the number of regions, but also minimize the redundant samples and hence, the monitoring overhead.

Chapter 7

Time-triggered Runtime Verification of Component-Based Multi-core Systems

Embedded systems interact with the physical world as well as execute on physical platforms. As embedded applications are increasingly being deployed on multi-core platforms, due to their inherent complex nature, the need for guaranteeing their correctness is further amplified. Consequently, it is essential to augment such systems with *runtime verification* technology. Our time-triggered monitoring presented in Chapters 4, 5, and 6 falls short in handling multi-core applications when several computing components execute concurrently.

In this chapter, we focus on extending the notion of time-triggered monitoring to the context of component-based multi-core embedded systems. The main challenge in this context is to identify the sampling period of the time-triggered monitors and a mapping from components and monitors to a set of computing cores, such that the cumulative monitoring overhead is *minimized*. To further describe this problem, consider a system with four components C_1, C_2, C_3, and C_4 that are executed only once. The system runs on two interconnected and identical computing cores P_1 and P_2, where each core hosts one time-triggered sampler. Each sampler has a fixed sampling period and samples all the components running on its host computing core. Table 7.1 shows the results of an experiment (see Subsection 7.3.3 for the settings) which measures the monitoring overhead (in milliseconds) imposed by a time-triggered sampler onto a component for different sampling periods. Details of the monitoring overhead will be thoroughly explained in the remainder of this chapter, but recall that in this thesis the monitoring overhead does not incorporate the overhead of the verification engine.

To demonstrate the importance of the mapping of components to computing cores, we randomly map the components in the following two ways:

1. $\{C_1, C_2, C_3\}$ runs on P_1 and $\{C_4\}$ runs on P_2.
2. $\{C_2, C_3, C_4\}$ runs on P_1 and $\{C_1\}$ runs on P_2.

In the first mapping, on P_1, the sampling period of 2 cycles and on P_2, the sampling period of 9 cycles achieve the optimal cumulative monitoring overhead (i.e., overall overhead on P_1 and P_2) which is 352ms. In the second mapping, on P_1, the sampling period of 7 cycles and on P_2, the sampling period of 2 cycles achieve the optimal overall monitoring overhead which is 283ms. As seen, the second mapping imposes 20% less monitoring overhead. This simple experiment indicates that the mapping of components to the computing cores affects the cumulative monitoring overhead and since embedded systems are usually resource constrained, it is highly desirable to find the mapping which results in the *least* cumulative monitoring overhead throughout the system run.

With this motivation, in this chapter, we propose an approach for optimizing the monitoring overhead of time-triggered monitoring in component-based multi-core embedded systems. That is, given a set of components, computing cores, and a set of allowed sampling periods, the goal is to identify:

1. the mapping of components to computing cores, and
2. the sampling period of each sampler,

such that the monitoring overhead of the time-triggered monitor is minimized, In other words, we aim at solving the following informal problem.

	Sampling Period [CPU cycles]								
	10	20	30	40	50	60	70	80	90
C_1	81	136	140	140	145	142	140	138	137
C_2	120	70	84	91	94	92	63	74	79
C_3	120	70	88	95	99	91	61	76	77
C_4	120	83	86	93	91	80	77	78	76

Table 7.1: Example of monitoring overhead [ms].

Problem statement. Given a set of components $\mathcal{C}$, a set of computing cores $\mathcal{P}$, where $|\mathcal{C}| \geq |\mathcal{P}|$, and a set of sampling periods SP, find the mapping of components in $\mathcal{C}$ to computing cores of $\mathcal{P}$, and sampling period s of each sampler, where $s \in SP$, such that the monitoring overhead of the time-triggered monitor is minimized.

To achieve this goal, in Section 7.1, we formalize the notion of monitoring overhead associated with terminating and non-terminating components [83]. Note that to this point, we considered the monitoring overhead as the time consumed to:

1. Invoke the time-triggered sampler,
2. Read the variables of interest, and
3. Stop and resume the inspected program's execution.

In this chapter, we incorporate more factors in calculating the monitoring overhead such as event buffering and execution of instrumentation instructions. Since the optimization problem is known to be intractable even for single component uni-core systems [19] (see Section 5.2), in Section 7.1, we introduce a mapping from our optimization problem to Integer Linear Programming (ILP) [83]. In order to incorporate the runtime characteristics of each component in our ILP model, we employ symbolic execution techniques [60].

Our approach is fully implemented within a tool chain and we report the results of rigorous experiments using the SNU [2] benchmark. In Secition 7.3, experimental results show that on average our approach can reduce the monitoring overhead of time-triggered monitoring by 34% as compared to various near-optimal monitoring patterns of the components at run time [83].

7.1 Optimal Monitoring of Component-based Systems

As previously discussed, the mapping of system components to the available computing cores can significantly affect the monitoring overhead imposed onto the system. Thus, we aim at finding the mapping that results in the *minimum* monitoring overhead. In this section, we present our target system architecture in Subsection 7.1.1, underlying objective in Subsection 7.1.2, the optimization problem, and its complexity analysis in Subsection 7.1.3.

7.1.1 System Architecture

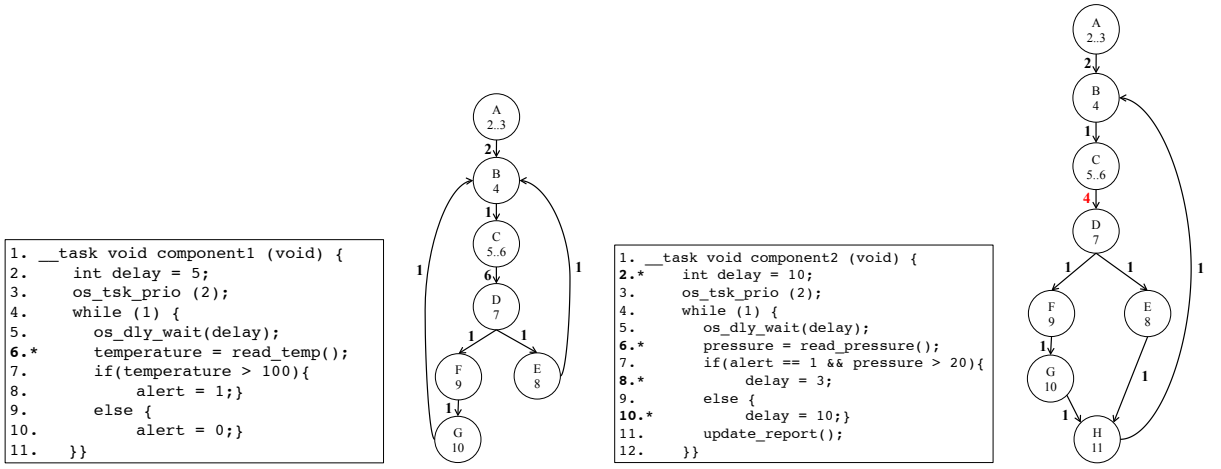
We follow an abstract view towards the underlying architecture of the system which is independent of the hardware, operating system, network protocol, etc. That is, a component-based system runs on a set of interconnected and potentially heterogeneous computing cores. However, we make the following assumptions:

- A component is either nonterminating or is invoked infinitely often throughout the system run.
- Given a set of components $\mathcal{C} = \{C_1 \cdots C_n\}$ and computing cores $\mathcal{P} = \{P_1 \cdots P_m\}$, where $m \leq n$, a function $\mathcal{F} : \mathcal{P} \rightarrow 2^{\mathcal{C}}$ maps each core in $\mathcal{P}$ to a unique subset of components of $\mathcal{C}$. Moreover, for any two distinct cores $P_1, P_2 \in \mathcal{P}$, we have $\mathcal{F}(P_1) \cap \mathcal{F}(P_2) = \{\}$. This function remains unchanged throughout the system execution; i.e., we are not concerned with dynamic scheduling of components.
- We assume a fully preemptive scheduler.
- The components can be invoked *aperiodically* throughout the system run.
- When a timing property involving a set of components requires verification, this set of components must run on the same core, so the property is soundly verified.
- We assume time-triggered samplers $\mathcal{S} = \{S_1 \cdots S_m\}$, where sampler S_i is deployed on core P_i. Each sampler observes and verifies *all* the components in $\mathcal{F}(P_i)$.
- No two components share a *variable of interest*; i.e., shared variables cannot be monitored.

7.1.2 Underlying Objective

The overhead imposed by the sampler is tightly coupled with the *execution path* of a component at run time (See Definition 9). When it is clear from the context, we abbreviate an execution path $\gamma = \langle (v_0, \omega_0, v_1), (v_1, \omega_1, v_2), ... \rangle$ by the sequence of its vertices $\gamma = \langle v_0, v_1, v_2, \ldots \rangle$. Moreover, for an infinite execution path γ, we represent the finite sub-execution path $\langle v_0, v_1, v_2, \ldots, v_n \rangle$ with γ^n, where $n \geq 0$.

For a finite path γ^n, monitored with a sampling period δ, we identify the following types of *time-related* overheads:



(a) Component1 (b) CFG of Component1 (c) Component2 (d) CFG of Component2

Figure 7.1: Component1 and Component2, along with their control-flow graphs

- $\mathcal{O}_c^\delta(\gamma^n)$ denotes the cumulative time spent for invoking the sampler throughout γ^n.
- $\mathcal{O}_r^\delta(\gamma^n)$ denotes the cumulative time spent for executing the monitoring code throughout γ^n (i.e., reading variables of interest and auxiliary memory).
- $\mathcal{O}_i^\delta(\gamma^n)$ denotes the cumulative time spent for executing the instrumentation instructions in γ^n. Recall that (from Subsection 5) instrumentation is used to increase longest sampling period LSP of a component.

Hence, the *time-related* monitoring overhead for γ^n is

$$\mathcal{O}_T^\delta(\gamma^n) = \mathcal{O}_c^\delta(\gamma^n) + \mathcal{O}_r^\delta(\gamma^n) + \mathcal{O}_i^\delta(\gamma^n)$$

Moreover, we identify the *memory-related* overhead, denoted by $\mathcal{O}_M^\delta(\gamma^n)$, which represents the auxiliary memory required to increase the longest sampling period LSP (see Subsection 5). To this end, we consider both the time-related and memory-related overheads to represent the monitoring overhead associated with time-triggered monitoring. Thus, we present the *monitoring overhead* as the pair $\mathcal{O}^\delta(\gamma^n) = \langle \mathcal{O}_T^\delta(\gamma^n), \mathcal{O}_M^\delta(\gamma^n) \rangle$.

Observe that the sampling period δ of a sampler considerably affects $\mathcal{O}^\delta(\gamma^n)$. That is, increasing δ results in decreasing the sampler invocations (i.e., $\mathcal{O}_c^\delta(\gamma^n)$), and increasing instru-

mentation instructions and the memory consumption (i.e., $\mathcal{O}_i^\delta(\gamma^n)$ and $\mathcal{O}_M^\delta(\gamma^n)$). For instance, in Component2 (see Figure 7.1(d)), consider execution path $\gamma_1 = \langle A, (B, C, D, E, H)^\omega \rangle$, where ω denotes the infinite execution of a sequence of basic blocks. Assuming that an instrumentation instruction takes 2 CPU cycles, for $\delta = 2$, where vertices E and G are instrumented, we have $\mathcal{O}_i^\delta(\gamma_1^n) = \frac{2n}{5}$. Note that for each sampling period δ, there is one and only one set of associated overheads and hence, a unique $\mathcal{O}^\delta(\gamma^n)$.

Clearly, for a finite path γ^n and two sampling periods δ_1 and δ_2, the time-related overhead incurred by δ_1 is better than the time-related overhead incurred by δ_2 iff

$$\mathcal{O}_T^{\delta_1}(\gamma^n) < \mathcal{O}_T^{\delta_2}(\gamma^n)$$

Accordingly, for an infinite path γ, the overhead incurred by δ_1 is better than the overhead incurred by δ_2 iff

$$\lim_{n\to\infty} \frac{\mathcal{O}_T^{\delta_1}(\gamma^n)}{\mathcal{O}_T^{\delta_2}(\gamma^n)} < 1 \tag{7.1}$$

For instance, in Component2 (see Figure 7.1(d)), execution path $\gamma_1 = \langle A, (B, C, D, E, H)^\omega \rangle$ and $\delta_1 = 2$, the time-triggered sampler is invoked $\frac{8n}{5\times 2}$ times where 8 is the best case execution time of $\langle B, C, D, E, H \rangle$. Assuming that the monitoring code and sampler invocation each take 5 CPU cycles, $\mathcal{O}_c^{\delta_1}(\gamma_1^n) = \mathcal{O}_r^{\delta_1}(\gamma_1^n) = 5 \times \frac{8n}{5\times 2}$. For $\delta_2 = 6$, where once again vertices E and G are instrumented, $\mathcal{O}_c^{\delta_2}(\gamma_1^n) = \mathcal{O}_r^{\delta_2}(\gamma_1^n) = 5 \times \frac{8n}{5\times 6}$, and $\mathcal{O}_i^{\delta_2}(\gamma_1^n) = \frac{2n}{5}$. As a result, δ_2 imposes less time-related overhead since:

$$\lim_{n\to\infty} \frac{\mathcal{O}_T^{\delta_2}(\gamma_1^n) = 2(5 \times \frac{8n}{5\times 6}) + \frac{2n}{5}}{\mathcal{O}_T^{\delta_1}(\gamma_1^n) = 2(5 \times \frac{8n}{5\times 2}) + \frac{2n}{5}} < 1$$

Our goal is to minimize the monitoring overhead associated with a component. Thus, for a finite set of possible sampling periods SP, and a component C with set of execution paths Γ, we want to find the sampling period $\Delta \in SP$ such that[1]:

$$\forall \delta \in SP : \lim_{n\to\infty} \frac{\mathcal{O}_T^{\Delta}(C) = \sum_{\gamma\in\Gamma} \mathcal{O}_T^{\Delta}(\gamma^n)}{\mathcal{O}_T^{\delta}(C) = \sum_{\gamma\in\Gamma} \mathcal{O}_T^{\delta}(\gamma^n)} \leq 1 \tag{7.2}$$

[1]Section 7.2.1 describes how to deal with finite paths by artificially making them infinite.

As discussed in Section 4.2.1, the internal structure of a component determines the longest sampling period and hence, affects the sampling period of the time-triggered sampler Δ. Moreover, the features of the computing cores define the set of possible sampling periods in SP which in practice is always finite.

In the general case, where a sampler S inspects a set of components $\mathcal{C}$, our underlying objective is to minimize the time-related overhead over all components. In other words, our objective is to identify Δ such that:

$$\forall \delta \in SP : \lim_{n \to \infty} \frac{\mathcal{O}_T^{\Delta}(S) = \sum_{C \in \mathcal{C}} \mathcal{O}_T^{\Delta}(C)}{\mathcal{O}_T^{\delta}(S) = \sum_{C \in \mathcal{C}} \mathcal{O}_T^{\delta}(C)} \leq 1 \tag{7.3}$$

One can develop the corresponding equations identical to Equations 7.1 – 7.3 for the memory-related overhead, and, hence, generalize these equations for the monitoring overhead.

7.1.3 Optimization Problem

In a system with multiple components that run on multiple computing cores, to minimize the *overall* monitoring overhead, in addition to calculating Δ (see Equation 7.3), one has to also identify an efficient mapping from components to cores. Thus, our optimization problem is as follows:

Problem statement. Given a set of components $\mathcal{C}$, a set of computing cores $\mathcal{P}$, where $|\mathcal{C}| \geq |\mathcal{P}|$, and a set of sampling periods SP, identify function $\mathcal{F} : \mathcal{P} \to 2^{\mathcal{C}}$ and sampling period Δ_P for each $\mathcal{F}(P)$, where $P \in \mathcal{P}$, such that

- the following is minimized:

$$\sum \{\mathcal{O}^{\Delta_P}(S_P) \mid P \in \mathcal{P}\}$$

- $\Delta_P \in SP$ (i.e., the sampling period of sampler S_P for components in $\mathcal{F}(P)$) satisfies Equation 7.3 with respect to the monitoring overhead,
- for any two computing cores $P_1, P_2 \in \mathcal{P}$, we have $\mathcal{F}(P_1) \cap \mathcal{F}(P_2) = \{\}$.

In Section 5.2, we show that optimizing the memory-related overhead and the sampling period even for one component is NP-complete. Thus, the NP-hardness of our optimization problem immediately follows. To tackle this obstacle, in Section 7.2, we propose a mapping from our optimization problem to integer linear programming.

7.2 Mapping to Integer Linear Programming

In this section, we introduce our mapping from the optimization problem in Subsection 7.1.3 to integer linear programming (ILP). Subsection 7.2.1 describes our technique to estimate the monitoring overhead associated with each component. Then, Subsection 7.2.2 presents the mapping to ILP.

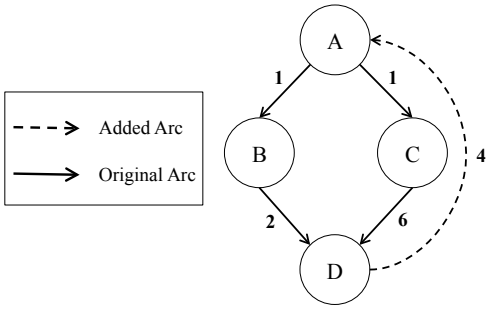
7.2.1 Calculating Overhead

To solve the optimization problem, we initially calculate the monitoring overhead associated with each component for every possible sampling period in SP. Recall that we focus on components that are nonterminating or are invoked infinitely often. Hence, for our analysis, we first convert all terminating components into nonterminating ones. To this end, for each component C and vertex v^t in the control-flow graph of C, where v^t has no outgoing arcs, we add an arc $\langle v^t, w_{v^t}, v^0 \rangle$ to the control-flow graph, where w_{v^t} is the best case execution time of v^t, and adjust C's source code accordingly. In other words, we encompass the complete body of the program under scrutiny in an infinite loop. We refer to the new component as *adjusted component*. The adjusted component is only used for pre-run analysis and does not replace the original component at run time. For instance, Figure 7.2(a) shows the control-flow graph of a terminating component with $v^0 = A$ and $v^t = D$. To adjust this component, we add arc $(D, w_D = 4, A)$ to its control-flow graph.

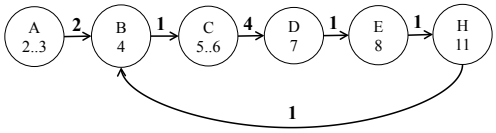
We now describe our approach for characterizing the monitoring overhead associated with a nonterminating component. Recall that to calculate the monitoring overhead, we require the set of all execution paths Γ of the component (see Equation 7.2). To properly calculate the monitoring overhead, Γ must satisfy the following requirement:

CFG Coverage The execution paths in Γ must completely cover the control-flow graph of the component. In other words, let CFG_γ be the control-flow graph covered by a path $\gamma \in \Gamma$. Then,

$$\bigcup_{\gamma \in \Gamma} CFG_\gamma = CFG$$



(a) Creating a nonterminating component.



(b) Sub-CFG of Component2.

Figure 7.2: Calculating monitoring overhead

Moreover, to calculate the monitoring overhead for each execution path $\gamma \in \Gamma$, we take the following steps. Consider path $\gamma = \langle A, (B, C, D, E, H)^{\omega} \rangle$ of Component2. To calculate its monitoring overhead, we extract a sub-execution path γ^n, such that the following requirements are met:

Path Coverage The length of γ^n should be long enough to cover the vertices in γ. This requirement is formalized as follows:

1. Let V_γ be the set of vertices that appear in γ. We require that for every vertex $v \in V_\gamma$, we have $v \in V_{\gamma^n}$. For instance, for any $n \geq 6$, $V_\gamma = V_{\gamma^n} = \{A, B, C, D, E, H\}$.
2. Let CFG_γ be the control-flow graph realized by execution path γ. We require that $CFG_\gamma = CFG_{\gamma^n}$. For instance, the set of vertices in γ and γ^{1000} induce the same control-flow graph in Figure 7.2(b). Observe that for path $\gamma^5 = \langle A, B, C, D, E, H \rangle$, $CFG_\gamma \neq CFG_{\gamma^5}$, since CFG_{γ^5} does not include arc (H, B).

Frequency Coverage The length of γ^n should be long enough such that the vertices which reside in an infinite loop are distinguishable from vertices that do not. This requirement is formalized as follows:

1. Let $f_\gamma(v)$ denote the number of times that vertex v appears in execution path γ. We require that for any two vertices v_1 and v_2 in V_γ, we have:

$$\left(\frac{f_\gamma(v_1)}{f_\gamma(v_2)} = \infty\right) \Rightarrow \left(\frac{f_{\gamma^n}(v_1)}{f_{\gamma^n}(v_2)} \approx n\right)$$

For instance, for vertices B and A in γ, if $n = 1000$, then

$$\frac{f_{\gamma^{1000}}(B)}{f_{\gamma^{1000}}(A)} = 994 \approx 1000$$

2. We also require that for two vertices v_1 and v_2 in V_γ, we have:

$$(\frac{f_\gamma(v_1)}{f_\gamma(v_2)} = k) \Rightarrow (\frac{f_{\gamma^n}(v_1)}{f_{\gamma^n}(v_2)} = k)$$

where k is a constant. For example, in path $\gamma' = \langle (A, B, C, B, C)^\omega \rangle$, for vertices A and B, $k = 2$. Thus, n has to be a multiple of 5. This coverage ensure that each vertex contributes the same ratio of execution time in γ^n as it does in γ.

By converting infinite execution paths to finite paths, the execution time of the paths may differ from one another. Hence, the monitoring overhead calculated for each path is based on a different scale (i.e., execution range). For example, consider paths γ_1^n and γ_2^n with best case execution time of 100 and 80 CPU cycles, and time-related overhead $\mathcal{O}_T^\delta$ of 50 CPU cycles. Evidently, in the long run, $\mathcal{O}_T^\delta(\gamma_2^n)$ will be larger than $\mathcal{O}_T^\delta(\gamma_1^n)$. Thus, to create comparable monitoring overheads, for each path γ^n, we *normalize* the monitoring overhead onto a scale of $[0, 1]$ as follows:

$$\mathcal{O}_\eta^\Delta(\gamma^n) = \frac{\mathcal{O}^\Delta(\gamma^n) - \mathcal{O}_{\min}(\gamma^n)}{\mathcal{O}_{\max}(\gamma^n) - \mathcal{O}_{\min}(\gamma^n)} \tag{7.4}$$

where $\mathcal{O}_{\min}$ is the least time-related overhead and $\mathcal{O}_{\max}$ is the largest time-related overhead that can be associated with γ^n. We consider $\mathcal{O}_{\min}(\gamma^n) = 0$ and $\mathcal{O}_{\max}(\gamma^n) =$ *best case execution time of* γ^n. We refer to $\mathcal{O}_\eta^\Delta$ as the *normalized monitoring overhead*.

Algorithm 5 calculates the normalized monitoring overhead $\mathcal{O}_\eta^\delta(C)$ associated with a component C as follows:

- For each sampling period δ, the algorithm optimally instruments C with respect to δ (Line 2), in the same fashion described in Chapter 5.
- Then, the algorithm extracts the set of finite execution paths Γ_{γ^n} of the instrumented component C' (Line 3). `get_paths` uses symbolic execution to extract the set of finite execution paths.

Algorithm 5 Calculating Normalized Monitoring Overhead

Input: SP: set of polling periods, C: a component
Output: $\Theta^C_{\mathcal{O}} = \{\mathcal{O}^{\delta_1}_{\eta}(C), \ldots, \mathcal{O}^{\delta_k}_{\eta}(C)\}$: set of monitoring overheads

1: **for** $\delta \in SP$ **do**
2: $C' \leftarrow opt\_instrument(C, \delta)$ /*instrument component*/
3: $\Gamma_{\gamma^n} \leftarrow get\_paths(C')$ /*get all finite execution paths*/
4: **for each** path $\gamma^n \in \Gamma_{\gamma^n}$ **do**
5: $\mathcal{O}^{\delta}_{i}(\gamma^n) \leftarrow 0$; $\mathcal{O}^{\delta}_{m}(\gamma^n) \leftarrow 0$; $\mathcal{O}^{\delta}_{\sum}(C) \leftarrow 0$
6: **for** vertex $v \in V_{\gamma^n}$ **do**
7: **if** v is instrumented **then**
8: /*adding instrumentation overhead*/
9: $\mathcal{O}^{\delta}_{i}(\gamma^n) \leftarrow \mathcal{O}^{\delta}_{i}(\gamma^n) + \Omega_{inst}(v)$
10: **end if**
11: **end for**
12: $\mathcal{O}^{\delta}_{M}(\gamma^n) \leftarrow get\_mem(\gamma^n, \delta)$
13: $\mathcal{O}^{\delta}_{T}(\gamma^n) \leftarrow \mathcal{O}^{\delta}_{i}(\gamma^n)$
14: $\mathcal{O}^{\delta}(\gamma^n) \leftarrow \langle \mathcal{O}^{\delta}_{T}(\gamma^n), \mathcal{O}^{\delta}_{M}(\gamma^n) \rangle$
15: $\mathcal{O}_{\max}(\gamma^n) \leftarrow$ BCET of γ^n
16: $\mathcal{O}^{\delta}_{\eta}(\gamma^n) \leftarrow \frac{\mathcal{O}^{\delta}(\gamma^n)}{\mathcal{O}_{\max}(\gamma^n)}$
17: $\mathcal{O}^{\delta}_{\sum}(C) \leftarrow \mathcal{O}^{\delta}_{\sum}(C) + \mathcal{O}^{\delta}_{\eta}(\gamma^n)$
18: **end for**
19: $\mathcal{O}^{\delta}_{\eta}(C) \leftarrow \frac{\mathcal{O}^{\delta}_{\sum}(C)}{|\mathcal{S}_{\gamma^n}|}$
20: $\Theta^C_{\mathcal{O}} \leftarrow \Theta^C_{\mathcal{O}} \cup \mathcal{O}^{\delta}_{\eta}(C)$ /*storing overhead for δ*/
21: **end for**
22: **return** $\Theta^C_{\mathcal{O}}$

- In Lines 7 – 10, for each path $\gamma^n \in \Gamma_{\gamma^n}$, the algorithm adds the overhead of each instrumentation instruction to $\mathcal{O}^{\delta}_{i}(\gamma^n)$. Function Ω_{inst} returns the best case execution time (in CPU cycles) of running the instrumentation.

- Then, Algorithm 5 calculates the memory-related overhead using function $get\_mem(\gamma^n, \delta)$ (Line 12). Note that at each sample, the monitor flushes out the auxiliary memory. Hence, get_mem creates a window of size δ and slides it through γ^n to find the maximum amount of data (in bytes) that a set of instrumentation residing in the window can store in the auxiliary memory, and considers it as the memory-related overhead. Algorithm 6 gives a detailed description of get_mem. Note that in Algorithm 6, at Line 5, $V_{\gamma^n}(v, \delta)$ extracts the list of vertices that reside in a window of

size δ and starting from vertex v.

- Next, the algorithm sets $\mathcal{O}_{\max}(\gamma^n)$ to the best case execution time of γ^n (Line 15).
- In Line 16, it calculates the normalized monitoring overhead $\mathcal{O}^{\delta}_{\eta}(\gamma^n)$ of γ^n and at line 17, it adds the normalized overhead to the overall monitoring overhead of the component $\mathcal{O}^{\delta}_{\sum}(C)$.
- Finally, the algorithm calculates the normalized monitoring overhead of the component $\mathcal{O}^{\delta}_{\eta}(C)$ by setting it to the average normalized monitoring overhead of the paths in Γ_{γ^n} (Line 19). Note that, since we do not know the exact frequency at which each path will be executed at run time, we consider the average normalized monitoring overhead among the paths. In the case where the probability distribution of the execution paths is known, we employ the *weighted* average.

Algorithm 6 get_mem

Input: γ^n: execution path, δ: sampling period
Output: $\mathcal{O}^{\delta}_{M}$: memory overhead

```
1:  m_MAX ← 0
2:  window ← create_window(δ) /*create window of size δ*/
3:  for vertex v ∈ V_γ^n do
4:     O_M^δ ← 0
5:     window ← V_γ^n(v, δ) /*vertices from v to the weight of δ*/
6:     for vertex υ ∈ window do
7:        if v is instrumented then
8:           O_M^δ ← O_M^δ + Ω_mem(v) /*adding memory consumption of instrumentation in v*/
9:        end if
10:    end for
11:    if m_MAX < O_M^δ then
12:       m_MAX ← Om
13:    else
14:       break
15:    end if
16: end for
17: return m_MAX
```

For a path γ, the sampler invocation overhead $\mathcal{O}^{\delta}_{c}(\gamma)$ depends on the number of times the sampler is invoked throughout the execution of γ which is tightly coupled with the

execution time of γ. Since statically computing the execution time of a path is unrealistic, it is equally impractical to estimate the number of sampler invocations. Thus, instead of calculating $\mathcal{O}_c^\delta(\gamma)$, we indirectly represent $\mathcal{O}_c^\delta(\gamma)$ using the value of the sampling period. It is straightforward to see that for a path γ, a larger sampling period results in less sampler invocations. Hence, for a component C and sampling periods δ_1 and δ_2, we have

$$(\frac{\delta_1}{\delta_2} = k) \Rightarrow (\frac{\mathcal{O}_c^{\delta_1}(C)}{\mathcal{O}_c^{\delta_2}(C)} = k)$$

where k is a constant. Thus, to minimize $\mathcal{O}_c^\delta(C)$, one should increase the sampling period δ. The overhead of running the monitoring code $\mathcal{O}_r^\delta(C)$ suffers from the same issue. We circumvent the problem in a similar fashion.

7.2.2 ILP Model

In order to cope with the exponential complexity of the optimization problem described in Subsection 7.1.3, we transform it into *Integer Linear Programming* (ILP). Our mapping takes as input the set of components $\mathcal{C}$, the set of samplers $\mathcal{S}$, the set of sampling periods SP, and the list of normalized monitoring overheads $\Theta_{\mathcal{O}}^c$ calculated by Algorithm 5 for each component $c \in \mathcal{C}$. In general, our objective is to minimize the cumulative normalized monitoring overhead over all the components in $\mathcal{C}$.

Variables. Our ILP model employs the following sets of variables:

1. $\mathbf{p} = \{p_m \mid m \in \mathcal{S}\}$, where each integer variable p_m has a value in SP, representing the sampling period of sampler m. This set of variables targets to find the optimal sampling periods Δ, as stated in the formal problem statement in Subsection 7.1.3. We emphasize that the solution to our ILP model gives the optimal sampling period Δ for each sampler, if overhead calculations are accurate. Section 7.3 discusses how we leverage symbolic execution for overhead calculation.

2. $\mathbf{x} = \{x_m^c \mid m \in \mathcal{S} \wedge c \in \mathcal{C}\}$, where each variable x_m^c represents the normalized monitoring overhead imposed on component c by sampler m. If m does not monitor c, then $x_m^c = 0$.

3. $\mathbf{y} = \{y_m^c, y'^c_m \mid m \in \mathcal{S} \wedge c \in \mathcal{C}\}$, where y_m^c and y'^c_m are called *choice variables*. The application of this set is described later in this section.

4. **ot** $= \{ot_m^c \mid m \in \mathcal{S} \wedge c \in \mathcal{C}\}$, where each variable ot_m^c presents the time-related overhead (i.e., $\mathcal{O}_T^\delta(c)$ calculated by Algorithm 5) imposed on component c when monitored by m.

5. **om** $= \{om_m^c \mid m \in \mathcal{S} \wedge c \in \mathcal{C}\}$, where each variable om_m^c presents the memory-related overhead (i.e., $\mathcal{O}_M^\delta(c)$ calculated by Algorithm 5) imposed on component c when monitored by m.

6. **z** $= \{z_m^\delta \mid m \in \mathcal{S} \wedge \delta \in SP\}$, where each z_m^k is a Boolean variable. The application of this set is described later in this section.

7. **ov** $= \{ov_m \mid m \in \mathcal{S}\}$, where each variable ov_m represents the cumulative normalized monitoring overhead imposed by m (i.e., $\mathcal{O}_\eta^{p_m}(m)$).

Constrains for sampling periods. For every sampler $m \in \mathcal{S}$, we add the following constraint to model all the possible values for a sampling period provided by SP:

$$s_m = \sum_{\delta \in SP} \delta \times z_m^\delta$$

Since s_m can have only one value from SP, we also add a constraint that *at most one* variable in $\{z_m^\delta\}_{\delta \in SP}$ can have a non-zero value.

Constrains for components. For each component $c \in \mathcal{C}$, we add the following constraints:

- *Constraint 1.* Each component $c \in \mathcal{C}$ must be monitored by one and only one sampler. Hence, variables $\{x_m^c\}_{m \in \mathcal{S}}$ are such that only *at most one* has a non-zero value. To this end, the set of variables $\{v_m^c\}_{m \in \mathcal{S}}$ reside in an *Special Ordered Set Type 1* (sos_1) (See Section 5.3). In addition, to guarantee that component c is indeed monitored, we have the following constraint:

$$\sum_{m \in \mathcal{S}} x_m^c \geq 1$$

- *Constraint 2.* When component c is monitored by a monitor m, x_m^c represents the normalized monitoring overhead for c, otherwise, $x_m^c = 0$. To model this behavior, we

employ choice variables y_m^c and ${y'}_m^c$. Variables y_m^c and ${y'}_m^c$ are such that one represents the normalized monitoring overhead and the other is zero. When m monitors c, $y_m^c > 0$ and ${y'}_m^c = 0$, otherwise, $y_m^c = 0$ and ${y'}_m^c > 0$. To this end, y_m^c and ${y'}_m^c$ reside in an *Special Ordered Set Type 1* (See Section 5.3). In addition, we have the following constraints for every monitor m:

$$x_m^c = y_m^c$$
$$y_m^c + {y'}_m^c = ot_m^c$$

As seen, y_m^c only represents the time-related overhead and not the memory-related overhead. This is because adding up ot_m^c and om_m^c that have different units (i.e., CPU cycles and bytes) is logically incorrect. Hence, we limit the memory-related overhead by defining an upper limit MEM.

$$\sum_{c \in \mathcal{C}} \sum_{m \in \mathcal{S}} om_m^c \leq MEM$$

We solve the optimization problem by running the model for different values of MEM and find the optimal solution.

- *Constraint 3.* For each sampler m, we calculate the time-related and memory-related overheads. Recall that $\Theta_{\mathcal{O}}^c = \{\mathcal{O}_\eta^\delta(c)\}_{\delta \in SP}$ is the set of normalized monitoring overheads where $\mathcal{O}_\eta^\delta(c) = \langle \mathcal{O}_T^\delta(c), \mathcal{O}_M^\delta(c) \rangle$ is calculated by Algorithm 5. To this end, we model all possible values of ot_m^c and om_m^c as follows:

$$ot_m^c = \sum_{\delta \in SP} \mathcal{O}_T^\delta(c) \times z_m^\delta, \text{ where } \mathcal{O}_T^\delta(c) \in \Theta_{\mathcal{O}}^c(\delta)$$
$$om_m^c = \sum_{\delta \in SP} \mathcal{O}_M^\delta(c) \times z_m^\delta, \text{ where } \mathcal{O}_M^\delta(c) \in \Theta_{\mathcal{O}}^c(\delta)$$

Constrains for samplers. The overhead imposed by a sampler m is the cumulative normalized monitoring overhead associated with all the components monitored by m. Hence, for each $m \in \mathcal{S}$, we have the following constraint:

$$ov_m = \sum_{c \in \mathcal{C}} x_m^c$$

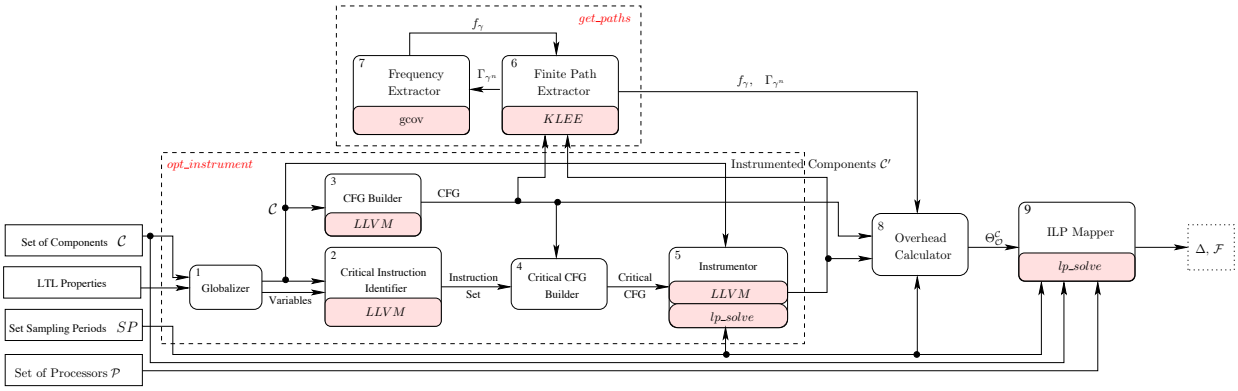


Figure 7.3: Tool chain

Optimization objective. In general, our objective is to map the components to samplers such that the cumulative normalized monitoring overhead over all samplers are minimized and also the sampling periods of the samplers is maximized. Thus, our ILP objectives are the following:

$$\min \sum_{m \in \mathcal{S}} ov_m$$

$$\sum_{m \in \mathcal{S}} s_m = MAX \tag{7.5}$$

where MAX is given as an input parameter to the ILP solver. Since off-the-shelf ILP solvers only support single objectives, we run the ILP model for all possible combinations of values for s_m by setting MAX from 1 to $|SP| \times |\mathcal{S}|$ to find the optimal solution.

7.3 Implementation and Experimental Results

In this section we present our tool chain in Subsection 7.3.1 and our experimental settings and the evaluation of our optimization approach in Subsections 7.3.2 and 7.3.3, respectively.

7.3.1 Implementation

Figure 7.3 presents the modules of our tool chain that implements our solution for solving the optimization problem described in Subsection 7.1.3.

opt_instrument. This module uses the ILP solution from Section 5.3 to optimally instrument the source code of the set of components $\mathcal{C}$ for each sampling period in the set SP. To this end, it takes the source code of each component in $\mathcal{C}$, a set of Linear Temporal Logic (LTL) properties, and the set of possible sampling periods SP. The *Globalizer* module extracts the set of variables of interest from the LTL properties and prepares the components in $\mathcal{C}$ for monitoring. In other words, *Globalizer* converts all the variables of interest in the code into global variables. The *CFG Builder* module extracts the control-flow graph of each component in $\mathcal{C}$ and provides it to *Critical Instruction Identifier*. The *Critical Instruction Identifier* module finds the set of critical instructions of each component in $\mathcal{C}$ with respect to the variables of interest. The *Critical CFG Builder* module uses the set of critical instructions and the set of control-flow graphs to create the set of critical control-flow graphs. Eventually, the *Instrumentor* module uses the set of critical control-flow graphs to optimally instrument the components in $\mathcal{C}$ for each sampling period in SP by leveraging the ILP solver lp_solve and LLVM [64] (see Section 5.3 for the ILP solution).

get_paths. This module extracts the set of finite execution paths of each instrumented component in $\mathcal{C}'$ (see Subsection 7.2.1). The *Finite Path Extractor (FPE)* module leverages the symbolic execution tool KLEE [24]. Finite Path Extractor initially *adjusts* the terminating components in $\mathcal{C}'$ (see Subsection 7.2.1) and runs KLEE to extract the set of execution paths Γ_{γ^n} of the components in $\mathcal{C}'$. Since the components in $\mathcal{C}'$ are nonterminating, we set an upper bound on the analysis time of KLEE. If all the paths in Γ_{γ^n} achieve path and CFG coverage (see Subsection 7.2.1), Finite Path Extractor sends Γ_{γ^n} to the *Frequency Extractor* module, otherwise, Finite Path Extractor increases the analysis time and restarts KLEE. When CFG coverage is not satisfied, Finite Path Extractor checks whether the uncovered control-flow graph is dead code. If so, Finite Path Extractor flags CFG coverage as unnecessary. Note that for an execution path, by increasing the analysis time of KLEE, we can potentially increase the length of the execution path. The *Frequency Extractor* module uses gcov to extract the execution frequency of each instruction of an execution path in Γ_{γ^n}. If the paths in Γ_{γ^n} do not satisfy frequency coverage (see Subsection 7.2.1), Finite Path Extractor increases the analysis time of KLEE. When the paths in Γ_{γ^n} satisfy all three coverages, Finite Path Extractor reports Γ_{γ^n} and the set of execution frequencies f_γ.

Overhead Calculator. This module implements Algorithm 5 using the characteristics of the computing cores and the overhead associated with each instrumentation, memory read and write, running the monitoring code, etc. This module calls opt_instrument (line 2 of Algorithm 5) and get_paths (line 3 of Algorithm 5) to calculate the normalized monitoring overhead.

ILP Mapper. This module solves the optimization problem using the ILP solution from Section 7.2.2 and returns the sampling period of each sampler (i.e., Δ) and the mapping of components to computing cores (i.e., $\mathcal{F}$). It uses lp_solve to solve the ILP model for all the possible combinations of sampling periods in SP (see Equation 7.5), and returns the solution which has the least monitoring overhead and largest sampling period.

7.3.2 Experimental Settings

We use two MCB1700 boards, Core1 and Core2, both running the RTX operating system. The time-triggered sampler on each board is a task with read access to all the variables of interest. At each sample the sampler writes the variables of interest and the auxiliary memory to an SD card for the verification engine (i.e., the actual monitor) to retrieve. The auxiliary memory on each board is 1600 bytes. We consider the following factors.

1. The mapping function $\mathcal{F}$.
2. The sampling period of each sampler.
3. The probability distribution for executing the components.

For evaluation, we run each experiment for one hour and measure the following metrics in 1-minute intervals (i.e., each experiment provides 60 data points):

1. The number of samples,
2. Overhead of invoking the time-triggered sampler $\mathcal{O}_c^\delta$,
3. Overhead of instrumentation $\mathcal{O}_i^\delta$,
4. Overhead of reading the variables of interest $\mathcal{O}_r^\delta$, and
5. Overhead of memory consumption $\mathcal{O}_M^\delta$.

Our case study leverages the SNU benchmark suite [2] to create a component-based embedded system. Each program is a component that is invoked infinitely often according to a normal distribution. Moreover, the set of possible sampling periods is $SP = \{5, \ldots, 30\}$.

7.3.3 Analysis of Experiments

We ran the ILP model for each possible combination of sampling periods used by the samplers running on the two boards. Figure 7.4(a) shows the ILP results (i.e., the optimal normalized monitoring overhead for each combination of sampling periods). Since the normalized monitoring overhead is a complex number (see Algorithm 5), in Figure 7.4(a), *normalized monitor-overhead* is the distance of the optimal normalized monitoring overhead to the origin of a complex plane. *Inverted sampling period*, in Figure 7.4(a), presents $\frac{1}{\sum_{m \in \mathcal{S}} s_m \times factor}$, where $\mathcal{S} = \{\mathsf{Core1}, \mathsf{Core2}\}$, s_m is the sampling period of the sampler on each core, and *factor* presents the impact that a sampler invocation along with monitoring code execution has on the monitoring overhead in comparison to an instrumentation. *factor* has a value of 100 in our experiments. Recall that a larger sampling period results in less runtime overhead (see Subsection 7.2.1). Figure 7.4(a) shows that the settings from point 20 is the solution to the optimization problem.

In addition, Row *Opt* in Table 7.2 presents the experimental results to the optimal solution from the ILP approach. Metric $\mathcal{F}$ is the mapping of components to cores and Δ is the optimal sampling period.

Impact of Mapping Function

We now evaluate the effectiveness of the mapping function of the optimal solution (i.e., Row *Opt* in Table 7.2). To this end, we change function $\mathcal{F}$ and recalculate Δ for each monitor using Equation 7.3. We create 13 different mappings, denoted by $M_{\mathcal{F}}$. Three mappings from $M_{\mathcal{F}}$ that have the closest monitoring overhead to *Opt* are shown in Table 7.2. Tables 7.3 and 7.4 show the experimental results for the mappings in Table 7.2. All the values are averages over 60 data points. Tables 7.3 and 7.4 represent the following information:

1. *Samples* is the number of monitoring samples.
2. $\mathcal{O}_T^{\Delta}\%$ is the percentage of the execution time consumed by the time-related overhead $\mathcal{O}_T^{\Delta}$.

Core1		
Setting	Metric	Settings
Opt	Δ	21 cycles
	$\mathcal{F}$	bs, jfd, ludcmp, sqrt, matmul, minver, qsort, insertsort, select
M_1	Δ	16 cycles
	$\mathcal{F}$	bs, crc, fft, fibcall, jfd, fir
M_2	Δ	16 cycles
	$\mathcal{F}$	bs, fft, lms, ludcmp, minver
M_3	Δ	16 cycles
	$\mathcal{F}$	bs, crc, fibcall, jfd, matmul, ludcmp
Core2		
Setting	Metric	Settings
Opt	Δ	23 cycles
	$\mathcal{F}$	crc, fibcall, fft, fir, lms, qurt
M_1	Δ	23 cycles
	$\mathcal{F}$	minver, lms, ludcmp, matmul, qsort, select, sqrt
M_2	Δ	23 cycles
	$\mathcal{F}$	crc, fibcall, fir, jfd, matmul, qurt, qsort, select, sqrt
M_3	Δ	23 cycles
	$\mathcal{F}$	fft, fir, lms, minver, qurt, qsort, select, sqrt

Table 7.2: Settings for Core1 and Core2

3. $\mathcal{O}_M^\Delta\%$ is the percentage of the consumed auxiliary memory $\mathcal{O}_M^\Delta$.

4. $\mathcal{O}^\Delta$ is the absolute value of the monitoring overhead (i.e., the distance to the origin of a complex plane), and

5. $\sigma_{\mathcal{O}_T^\Delta}$, $\sigma_{\mathcal{O}_M^\Delta}$, and $\sigma_{\mathcal{O}^\Delta}$ are the standard deviations of $\mathcal{O}_T^\Delta$, $\mathcal{O}_M^\Delta$, and $\mathcal{O}^\Delta$, respectively.

Table 7.4 shows that on average *Opt* imposes 20.46% less monitoring overhead in comparison to M_1–M_3. Table 7.3 shows that *Opt* imposes 11.86% more memory-related overhead $\mathcal{O}_M^\Delta$ in comparison to M_1. On the other hand, M_1 imposes 19.98% more time-related overhead $\mathcal{O}_T^\Delta$. Although, M_1 imposes less $\mathcal{O}_M^\Delta$, the impact of $\mathcal{O}_T^\Delta$ is stronger on $\mathcal{O}^\Delta$ (i.e., *factor* is 100 in our experiments). This observation matches with the observations in [19]. The larger $\mathcal{O}_T^\Delta$ of M_1–M_3 is caused by the larger number of samples which is the outcome of the smaller sampling period on Core1. Moreover, the experiments on all the mappings

Setting	Sample	$\mathcal{O}_c^\Delta$ [ms]	$\mathcal{O}_i^\Delta$ [ms]	$\mathcal{O}_r^\Delta$ [ms]	$\mathcal{O}_T^\Delta$ [ms]	$\mathcal{O}_M^\Delta$ [byte]
Opt	250,617.98	1,700.69	6.26	2,607.79	4,314.75	925.33
M_1	361,275.83	2,443.04	5.82	2,727.42	5,176.29	827.2
M_2	360,929.61	2,441.17	7.74	2,744.72	5,193.64	984
M_3	368,088.71	2,493.52	9.07	2,828.90	5,331.50	1,004.26

Table 7.3: Monitoring overhead of SNU programs [First set of data]

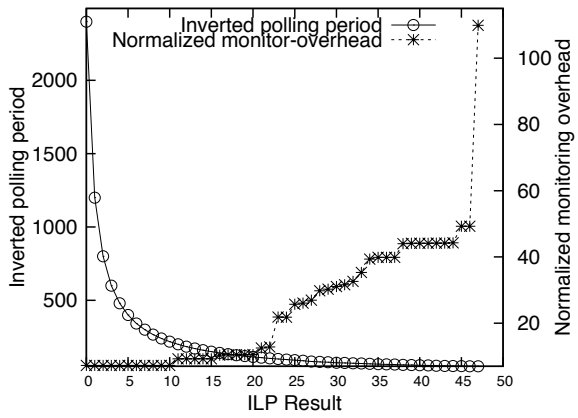
Setting	Sample	$\mathcal{O}_T^\Delta\%$	$\mathcal{O}_M^\Delta\%$	$\sigma_{\mathcal{O}_T^\Delta}$	$\sigma_{\mathcal{O}_M^\Delta}$	$\mathcal{O}^\Delta$ [ms]	$\sigma_{\mathcal{O}^\Delta}$
Opt	250,617.98	7.02	57.83	4.75	24.73	4,412.84	12.30
M_1	361,275.83	8.62	51.7	4.48	28.67	5,242.58	10.81
M_2	360,929.61	8.65	61.5	4.68	27.29	5,279.62	13.28
M_3	368,088.71	8.88	62.73	5.13	31.83	5,425.21	12.84

Table 7.4: Monitoring overhead of SNU programs [Second set of data]

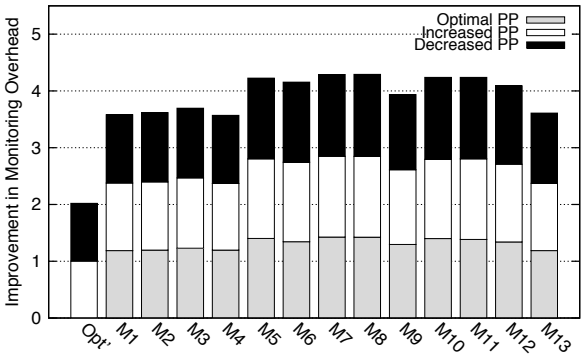
in $M_\mathcal{F}$ show that on average *Opt* imposes 31.1% less monitoring overhead. *Optimal-PP*, in Figure 7.4(b), shows the ratio of the monitoring overhead associated with each mapping in $M_\mathcal{F}$ to the monitoring overhead associated with *Opt*. In addition, the results show that our optimization approach has an error factor of 0.07. In other words, in one out of 13 mappings, our approach did not accurately reflect the monitoring overhead due to the normalization in Equation 7.4. The significance test on all these experiments show that the results are statistically significant with a p-value of less than 1^{-100}, hence, our approach can successfully find the mapping that results in the minimum monitoring overhead.

Impact of sampling Period

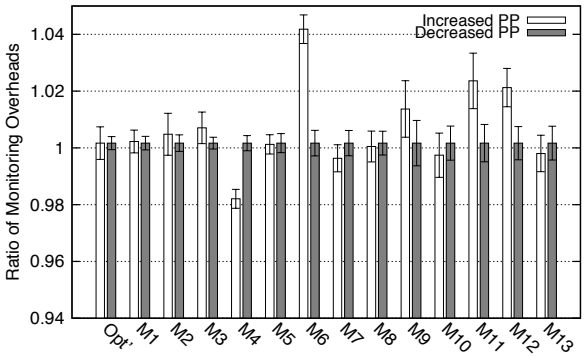
We change the sampling period of the sampler on each core for every mapping in $M_\mathcal{F}$ (i.e., the thirteen mappings). We once increase and once decrease the sampling period of each sampler by 2 CPU cycles. Figure 7.4(b) shows the ratio of the monitoring overhead associated with each mapping in $M_\mathcal{F}$ to the monitoring overhead associated with *Opt* with respect to the changed sampling periods. On average *Opt* imposes 32.81% less monitoring overhead in comparison to the monitoring overhead associated with the mappings in $M_\mathcal{F}$. Moreover, in Figure 7.4(c), for each mapping M, *increased/decreased-PP* presents the ratio of the monitoring overhead associated with mapping M when using the increased/decreased sampling periods to the monitoring overhead of mapping M when using the optimal sampling period Δ associated with M. The error bars reflect the standard deviation. The results show that by employing Δ, the mappings in $M_\mathcal{F}$ impose 1.39% less



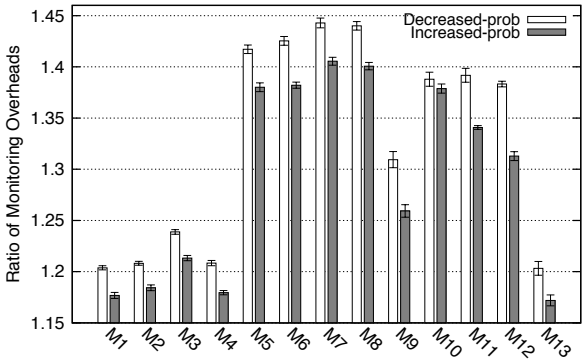
(a) ILP results for SNU.



(b) Factor: mapping function.



(c) Factor: sampling period.



(d) Factor: probability distribution.

Figure 7.4: Experimental results

monitoring overhead. The error factor of this set of experiments is 0.19; i.e., in 5 out of 26 experiments, our approach did not correctly calculate Δ. The significance test on all these experiments show that the results are statistically significant with a p-value of less than 1^{-100}.

Robustness Analysis

To calculate the normalize monitoring overhead, we assumed that the execution paths of each component and the set of components are executed based on a normal distribution. We change the probability distribution as follows to evaluate the robustness of our optimization approach:

1. We increase the probability distribution of the execution path(s) and component(s) with the highest normalized monitoring overhead, by 10 units. In Figure 7.4(d), *increased-prob* shows the ratio of the monitoring overhead associated with each mapping in $M_{\mathcal{F}}$ to the monitoring overhead associated with *Opt* with respect to the new probability distribution. The error bars reflect the standard deviation. Figure 7.4(d) shows that with moderate increases in the probability distribution, our approach is still effective since on average *Opt* imposes 33.14% less monitoring overhead. By also changing the sampling period as in Subsection 7.3.3, on average *Opt* imposes 34.1% less overhead. The experiments show an error factor of 0.38, meaning that in 15 out of 39 experiments, our approach did not either correctly reflect the monitoring overhead or calculate the optimal sampling period.

2. Likewise, we decrease the probability distribution of the execution path(s) and component(s). Figure 7.4(d) shows that with moderate decreases in the probability distribution, our approach is still effective since on average *Opt* imposes 27.27% less monitoring overhead. By also changing the sampling period as in Subsection 7.3.3, on average *Opt* imposes 29.2% less overhead. The experiments show an error factor of 0.25, meaning that in 10 out of 39 experiments, our approach did not either correctly reflect the monitoring overhead or calculate the optimal sampling period.

We note that the above error factors are expected, since our overhead calculations are based on normal distribution. Having said that, these experiments shows that changing the probability distribution does not significantly undermine the robustness of our approach. Thus, the insight is when the probability distribution of the components are approximately known, it is advisable to use the knowledge when calculating the normalized monitoring overheads in Algorithm 5. In addition, in all the aforementioned experiments, the significance test show that the results are statistically significant with a p-value of less than 1^{-100}.

Chapter 8

RiTHM: A Tool for Enabling Time-triggered Runtime Verification for C Programs

In Chapters 4 and 5, we established the theoretical basis for time-trigged monitoring for real-time embedded systems. Recall that we required a runtime verification framework that satisfies the following properties.

1. Bounded overhead,
2. Predictable monitoring,
3. Efficient monitoring, and
4. Reduced over-provisioning.

Experimental results from Chapter 4 prove that our hypothesis regarding the above runtime properties truly hold. As for efficient monitoring, in Chapter 5, we presented the use of history to reduce the monitoring overhead of time-triggered monitoring. Experiments showed that by using history, time-triggered monitoring can result in efficient monitoring while still providing bounded overhead, predictable monitoring, and reduce over provisioning. To this end, after carrying out detailed studies on time-triggered runtime verification, we gathered all the aspects of our time-triggered monitor which have been thoroughly investigated via rigorous testing into a stand alone tool.

In this chapter, we introduce the tool RiTHM [84]: Runtime Time-triggered Heterogeneous Monitoring. RiTHM combines the work in Chapters 4 and 5 alongside additional features which we will present in the remaining of this chapter. Despite the long history of runtime monitoring, we know of no tools that enable runtime monitoring of real-time embedded systems.

In general, RiTHM takes a C program under inspection and a set of Ltl properties as input and generates an instrumented C program that is verified at run time by a *time-triggered* monitor. The current implementation of RiTHM supports two time-triggered monitoring and instrumentation techniques:

1. Time-triggered monitoring with optimized *fixed* sampling period using static analysis [19], and
2. Time-triggered monitoring with least variation in dynamic sampling period using PID and fuzzy controllers [73].

The verification decision procedure (i.e., the monitor itself) for 3-valued semantics of Ltl [9] is sound and complete [19], and takes advantage of the GPU many-core technology to speedup monitoring and isolating the monitoring tasks (i.e., the verification procedure) [13] [11]. The tool has been used in several real-world case studies such as the Apache web server, a UAV autopilot software, and a laser-beam stabilizer for eye surgery.

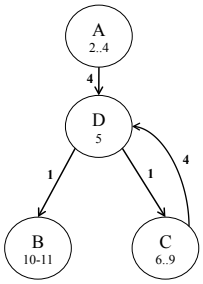
8.1 Tool Overview

Figure 8.11[1] shows modules and detailed data flow of RiTHM. The tool takes a C program and a set of Ltl properties as input and generates an *instrumented* C program which can be monitored and verified by a time-triggered monitor as output. The specification language for expressing the input properties is the 3-valued Ltl designed particularly for runtime verification [9]. Each given Ltl property is specified in terms of variables of the input C program. For instance, $\mathsf{G}(\mathsf{x} > 10 \text{ and } \mathsf{foo\_y} = \mathsf{z})$ is one such property, where x and z are two global variables and y is local to function foo. Note that local variables are presented with respect to their context (i.e., defining functions). With this respect, two variables with the same identifier defined in different contexts can be distinguished from one another.

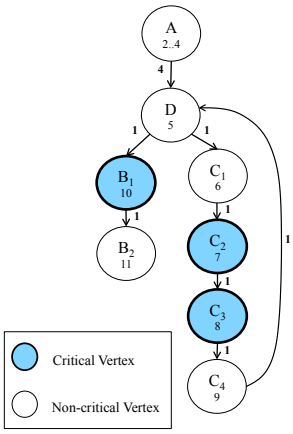
[1]Note that, in this chapter to eliminate confusion between the sampling period of the time-triggered monitor and the sampling period of the controllers in Section 8.1.2, we refer to the sampling period of the time-triggered monitor as *polling period* in the figures of this chapter.

```
1. fib(int n) {
2.     int  i, Fnew, Fold, temp,ans;
3.     Fnew = 1;  Fold = 0;
4.     i = 2;
5.     while( i <= n ) {
6.         temp = Fnew;
7.*        Fnew = Fnew + Fold;
8.*        Fold = temp;
9.         i++; }
10.*   ans = Fnew;
11.    return ans;}
```

(a) Fibonacci function in C



(b) Control-flow graph



(c) Critical control-flow graph

Figure 8.1: Example of a program

RiTHM uses the following modules to create the final instrumented C program that is ready to be monitored by a time-triggered monitor.

- *Globalizer*: The intuition behind Globalizer (Module 1 in Figure 8.11) stems from the method the time-triggered monitor uses to extract the variables of interest. Recall that the time-triggered monitor extracts all the variables of interest at each sample. Consequently, in the case where more than one local variable of interest with the same identifier exist, the time-triggered monitor can not differentiate them from one another at a sample point. To this end, Globalizer renames the local variables of interest by prefixing the identifier of their context (i.e., function) to the variable's

identifier. For instance, for local variable x defined in context foo, Globalizer renames variable x to foo_x. *Globalizer* is implemented over LLVM Clang [64]. It takes the C program and LTL properties as input and generates a C program where all the local variables participating in the LTL properties are renamed and changed into global variables. Globalizer generates the list of the globalized variables and passes it to LTL_3 *Monitor Generator*.

- LTL_3 *Monitor Generator*: The intuition behind the Monitor Generator (Module 3 in Figure 8.11) stems from the need for an engine to verify the set of LTL properties at run time (i.e., the monitor itself). To this end, Monitor Generator develops a kernel code for GPU multi-code to carry out the algorithms to verify the set of LTL input properties. Section 8.1.3 thoroughly explains the LTL evaluation procedure.

- *Critical Instruction Identifier*: This module (Module 4 in Figure 8.11) identifies and annotates the set of the C program's instructions which may change the valuation of the properties at run time. In other words, this module is a static analysis technique which runs over LLVM. It takes the internal representation of the program provided by LLVM and parses the program to extract the set of instructions that change the value of the variables of interest. This module leverages alias analysis provided by LLVM to improve its precision.

At this point, RiTHM gives two options for generating a time-triggered monitor, as described in Subsections 8.1.1 and 8.1.2. Regardless of the sampling technique, RiTHM uses a unified verification engine, as described in Subsection 8.1.3.

8.1.1 Instrumentation for Time-triggered Monitoring with Fixed Sampling Period

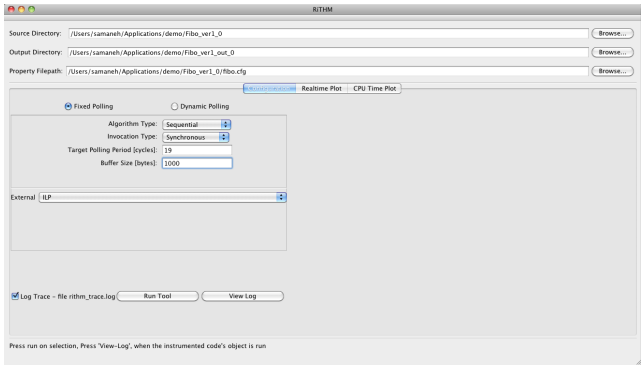
The first tool option for enabling time-triggered runtime verification is to augment the input program with a time-triggered monitor that employs a given fixed sampling period. This option incorporates the work presented in Chapters 4 and 5. To this end, RiTHM uses the following modules to create a time-triggered monitor that uses a fixed sampling period to extract the variables of interest.

- *CFG Builder*: Globalizer sends the modified C program to CFG Builder (module 2 in Figure 8.11) that generates the control-flow graph of the program. This module is implemented over LLVM and leverages its dot_cfg analysis pass. CFG Builder uses

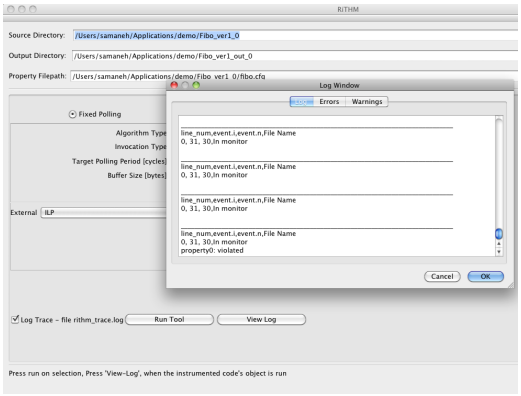
information regarding the underlying hardware which will run the program and the time-triggered monitor to calculate the best case execution time of each instruction of the program. Note that hardware vendors provide a manual which incorporates the best case execution time of each assembly code instruction in the form of CPU cycles. Eventually, CFG Builder uses the timing information to create the control-flow graph of the program presented in Definition 1. Figure 8.1(b) shows the control-flow graph created by CFG Builder for the Fibonacci function in Figure 8.1(a). The weight of each arc is the best-case execution time of the instructions in the originating vertex. For simplicity, in this example, we assume that each instruction takes one time unit.

- *Critical CFG Builder*: This module (module 5 in Figure 8.11) constructs a critical control-flow graph, where each critical instruction resides in one and only one vertex which conforms to the definition of critical control-flow graph in Section 4.2.1. Figure 8.1(c) shows the critical control-flow graph of Figure 8.1(a), where variables Fnew, Fold, and ans appear in the input LTL properties (i.e., are variables of interest).

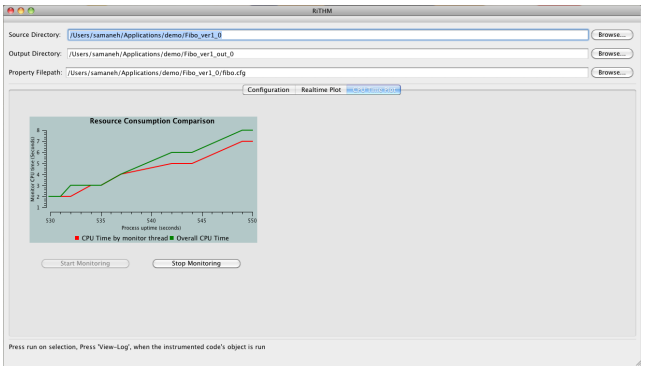
- *Optimization Module*: This module (module 6 in Figure 8.11) takes a desired sampling period and the critical control-flow graph as input and uses three optimization techniques to instrument the C program to ensure sound runtime verification (see Chapter 5). Recall that in order to ensure sound verification, the sampling period should be not be greater than the shortest time between the execution time of two critical instructions (i.e., longest sampling period, *LSP*). If the input sampling period is greater than *LSP*, then some critical instructions must be instrumented, so that their results are temporarily stored in a *history* buffer until the sampler's next sample (see Chapter 5). In terms of a control-flow graph, instrumenting a critical instruction involves deleting its corresponding vertex in the critical control-flow graph and merging outgoing and incoming arcs of the vertex by summing up their pairwise weights. Recall from Section 5.2, we require that the number of instrumentations to be minimum and have shown that the corresponding optimization problem is NP-complete. To this end, RiTHM brings together the following optimization techniques (see Sections 5.3 and 5.4) to solve the optimization problem: (1) integer linear programming (ILP) (see Section 5.3), (2) a greedy heuristic (see Section 5.4), and (3) a heuristic based on finding the minimum vertex cover (see Section 5.4). The output of either technique is a set of instructions in the C program that need to be instrumented. For the program in Figure 8.1(a), to achieve a sampling period of 2 time units, the ILP solution for the critical control-flow graph in Figure 8.1(c) instruments vertices C_2 (Line 7) and B_1 (Line 10). The lines of code corresponding to these instructions are located using the abstract syntax tree generator (module 8 in Figure 8.11) of LLVM



(a) Configuration of time-triggered monitor with fixed sampling period



(b) Log file from RiTHM using fixed sampling period



(c) CPU consumption of time-triggered monitor with fixed sampling period

Figure 8.2: Selected RiTHM screen shots for fixed sampling period

Clang, and instrumented by *Instrumentor 1* (module 9 in Figure 8.11).

- *Glue Code Generator 1*: This module (module 13 in Figure 8.11) augments the instrumented C program with function calls to the verification engine created by LTL_3 Monitor Generator for verifying the LTL properties at run time.

Figure 8.2(a) shows the screen shot of configuring RiTHM to generate this type of time-triggered monitor. The *Algorithm type* and *Invocation type* refer to the settings for the verification engine which we will discuss in Section 8.1.3. The *Target polling period* represents the desired sampling period for the time-triggered monitor. The *Buffer size*

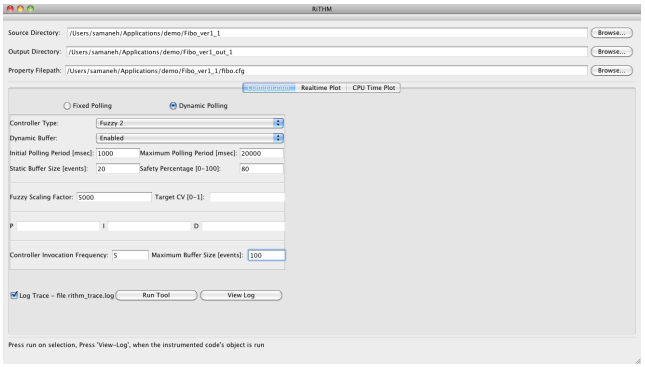
represents the size of history used to store variables of interest. *External* represents the type of optimization technique to be used for instrumentation. The options are ILP, Greedy, and VC. Figure 8.2(b) shows the log file from the verification engine. As seen, RiTHM tracks and stores the values of variables of interest and the evaluation of the LTL properties at each sampling point. With this respect, the developer can use the log to analyze the runtime behavior of the program offline. In addition, Figure 8.2(c) shows how RiTHM tracks the CPU consumption of both the time-triggered monitor and the program + time-triggered monitor combined, at run time.

8.1.2 Instrumentation for Time-triggered Monitor with Dynamic Sampling Period

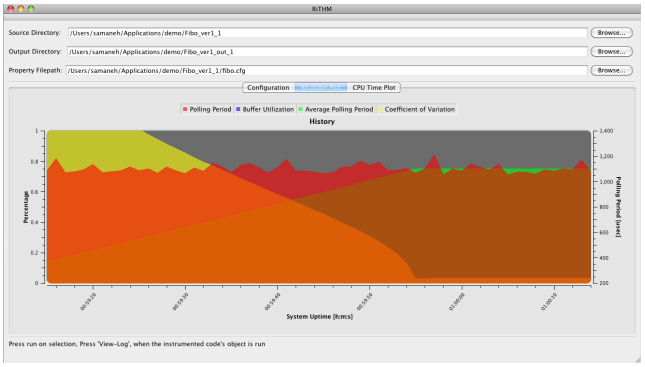
Since in reactive systems, environment events play an important role in determining the sampling period required to achieve sound verification, techniques based on static analysis presented in Chapters 4, 5, and 6 are not expected to be effective. To deal with reactive systems, RiTHM can also generate time-triggered monitor's augmented with PID and fuzzy controllers (module 12 in Figure 8.11) that can dynamically change the sampling period of the time-triggered monitor based on the environment behavior. Specifically, given the range of allowed sampling periods and constraints on the static/dynamic history buffer size, the controllers target minimizing variations in adjustments to the dynamic sampling period as well as maximizing history buffer utilization. Instrumentation and controller-based time-triggered monitor generation are achieved through modules 10 and 11, respectively [73].

Figure 8.3(a) shows the screen shot for configuring controller-based time-triggered monitors. The *Controller type* represents the type of controller to be used. The options are PID, Fuzzy 1, Fuzzy 2, Fuzzy 3, Fuzzy 3 - remapped. The fuzzy controllers differ in their fuzzy scaling and goals. The *Dynamic buffer* represents if the time-triggered monitor should dynamically increase the buffer size used by the controllers at run time. *Initial polling period* represents the starting sampling period of the time-triggered monitor and *Maximum polling period* represents the upper limit for acceptable sampling periods for the time-triggered monitor. *Static buffer size* presents the size of the buffer used by the verification engine. *Safety percentage* presents the lower limit on the accuracy percentage of the controllers. *Fuzzy scaling factor* and *Target CV* are specialized settings to customize the controllers. *Controller invocation Frequency* presents the time interval to invoke the controller at run time and *Maximum buffer size* sets an upper limit on the buffer size used by the controllers.

RiTHM provides a real-time plot of the current sampling period and the utilization of the available buffer as seen in Figure 8.3(b). The red curve presents the changes made



(a) Configuration of time-triggered monitor with dynamic sampling period



(b) Realtime plot for RiTHM's dynamic sampling period

Figure 8.3: Selected RiTHM screen shots for dynamic sampling period.

to the time-triggered monitor's sampling period. The yellow curve presents the coefficient variation of the time-triggered monitor's sampling period, and the green curve presents the average sampling period of the time-triggered monitor.

8.1.3 Verification Engine

Evaluation of LTL properties at run time is handled by a GPU-based verification engine (module 3 in Figure 8.11). If a machine is not equipped with the GPU, the verification is automatically shipped to a multi-core CPU. Custom code can be included in the monitor thread of the tool. For example, one can customize the verification engine to verify only a subset of properties. The verification engine is invoked by the time-triggered sampler

thread that RiTHM generates using Unix high resolution timers (module 7 in Figure 8.11). The sampler stops the C program's execution with a fixed/dynamic polling period, reads the program state, sends the extracted data to the verification engine, and resumes the program thread. The verification engine evaluates properties in parallel with the program execution.

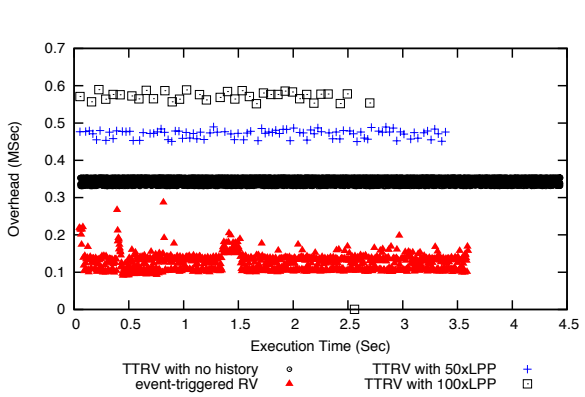
The current version of the verification process implements a multi-threaded (and not multi-process) architecture. The verification process might be either blocking (i.e., synchronous) where the main thread invokes the flush function and waits until it is done or non-blocking (i.e., asynchronous) where the main thread designates a worker thread from the pool to perform the verification.

Currently, the verification engine has four different algorithms for the properties verification. They can be listed in the declining order of the CPU engagement: Sequential, Partial Offload, Finite-History and Infinite-History algorithm. For detailed descriptions of these algorithms visit the verification engines official website https://bitbucket.org/sberkovich/goomf/wiki/Home.

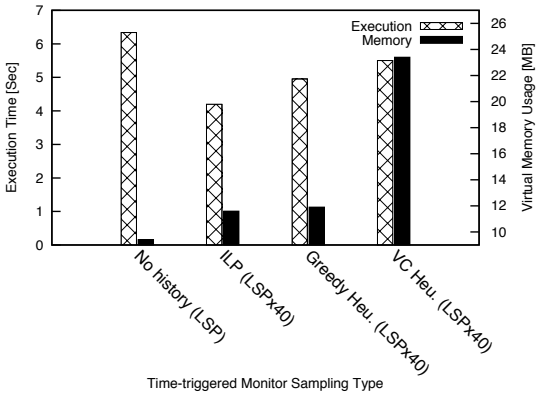
8.2 Selected Experiments

Figure 8.4(a) [19] shows that the absolute overhead incurred by RiTHM when using fixed sampling period (with and without history) is bounded and uniform and hence, predictable, as opposed to an event-triggered monitoring. The event-triggered monitor is implemented by a function that is invoked by each event that has to be monitored. Notice that the choice of event-trigger is irrelevant. Figure 8.4(a) shows that the execution time of the program monitored by RiTHM with fixed sampling period without history is larger than the execution time of the program monitored by event-triggered monitoring. This observation fits the results seen in Chapter 4. However, by extending the sampling period (e.g., by a factor of 100), RiTHM performs better than event-triggered monitoring. Figure 8.4(b) shows the execution time and memory usage of the program blowfish when instrumented by ILP and the other RiTHM instrumentation heuristics for the sampling period of $40 \times LSP$.

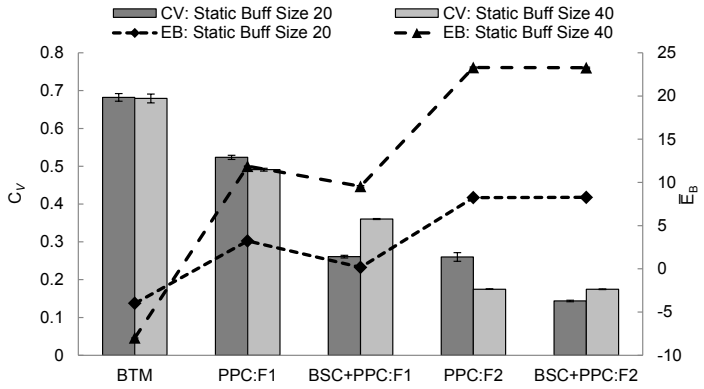
Figure 8.4(c) shows the coefficient of variation (CV) of the time-triggered monitor's sampling period and memory utilization (in terms of the number of empty locations in the history buffer, where positive is excessive dynamic allocation and negative denotes an under utilized buffer) for the Apache web server using a controller-based time-triggered monitor. As can be seen, the monitor that is controlled by two fuzzy controllers for stabilizing the time-triggered monitor's sampling period and buffer size (i.e., BSC+PPC:F2) shows



(a) TTRV of blowfish.



(b) Memory vs. execution time.

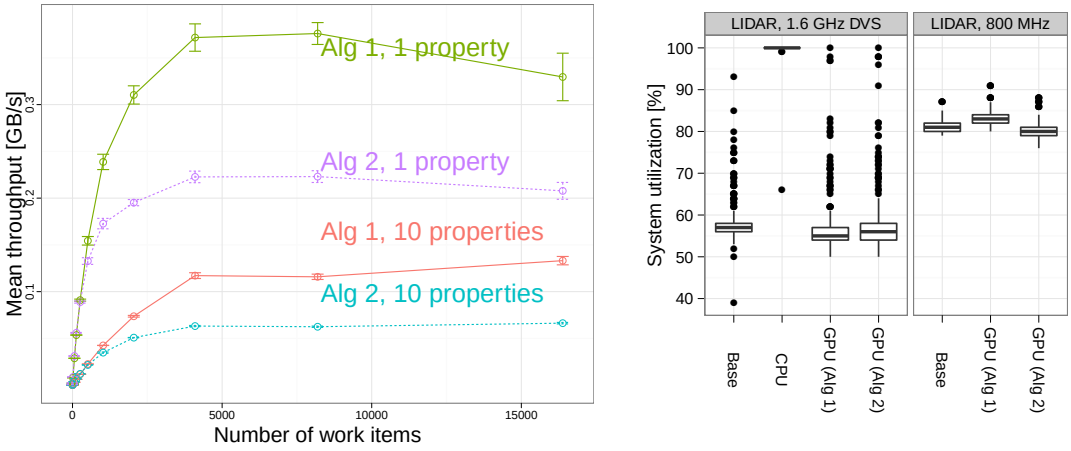


(c) Controller-based TTM.

Figure 8.4: Selected experiments.

a significantly low CV and well-utilized history buffer. The data set of this experiment is from the 1998 FIFA World Cup web server.

Figure 8.5(a) shows that our GPU-based algorithms for verifying LTL_3 properties are clearly scalable with respect to the number of cores. The error bars represent a 95% confidence interval. This graph also shows that the mean throughput increases with the number of cores engaged in monitoring. At some point, the parallel verification algorithms reach the optimum, where all the cores are utilized. Figure 8.5(b) shows that the CPU utilization of the autopilot process of an unmanned aerial vehicle (UAV) application monitored using our GPU-based algorithms is almost identical to the CPU utilization of the unmonitored process. Notice that the same program monitored by a CPU-based monitor is



(a) Scalability.

(b) CPU utilization.

Figure 8.5: Selected GPU-based verification experiments.

almost 100% utilized. This result holds when the CPU frequency is reduced to half of the normal frequency. An interesting side-effect of this result is that GPU-based monitoring is considerably more power-efficient.

8.3 Limitations of RiTHM

The fixed sampling period functionality of RiTHM (i.e., the cumulative methods present in this thesis) currently suffers from a number of technical limitations as listed below:

- Fixed sampling period can only consider primitive data types as variables of interest (i.e., variables used in the properties of interest that are to be verified).
- Fixed sampling period's accuracy in calculating the longest sampling period is limited to the accuracy of LLVM's underlying alias analysis. In other words, if a pointer is a variable of interest, RiTHM uses alias analysis to find all the critical instructions that update the value of the pointer. Since none of the currently practical alias analysis techniques are sound and complete, RiTHM's fixed sampling period calculation suffers from inaccuracy.
- Fixed sampling period can not monitor dynamically allocated memory. To this end, dynamically allocated memory can not be used in properties of interest

- Fixed sampling period can not soundly calculate the longest sampling period in multi-threaded programs. In other words, the method presented in this thesis can not calculate the longest sampling period when a variable of interest is updated by multiple threads.
- The integer linear programming solver, lp_solve, can not scale to large programs with a control-flow graph of tens of thousands of edges and vertices.
- RiTHM can only run on x86 or ARM-Cortex-M3.
- RiTHM can only run on Unix based systems and ARM RTX.

8.4 Availability

RiTHM is an open source tool. To access the tool, related publications, screencasts, more detailed experimental results, user guide, and other resources, please visit http://uwaterloo.ca/embedded-software-group/projects/rithm.

8.5 The Demo

The demo illustrates RiTHM by designing, enabling monitoring, and executing a simple Fibonacci example for the fixed sampling period, and dynamic sampling period. The demo consists of four parts:

1. application design,
2. development of LTL properties,
3. applying RiTHM to synthesize monitors and instrument programs, and
4. demonstrating RiTHM's features on collecting statistics about satisfaction/violation of properties, trace logs, and timing behavior of monitors.

8.5.1 Application Design

We develop the Fibonacci function as illustrated in Figure 8.6(a) as a C program with a given input value n. The demo will analyze the corresponding control-flow graph of Fibonacci (see Figure 8.1(b)).

8.5.2 Specification Development

We demonstrate the monitoring of three properties (see Figure 8.6(b)):

- The first property property0 states that the loop counter eventually reaches a value greater than 10.
- The second property property1 states that it is always the case that the loop counter i is less than or equal to n.
- The third property property2 intends to monitor that an element in the output Fibonacci series is strictly greater than the previous element.

8.5.3 Instrumentation Phase

The RiTHM configuration screenshot is shown in Figure 8.2(a). Running RiTHM results in the log shown Figure 8.7. This window also shows possible errors and warnings. For instance, an error could be using a variable in a property that is never declared. A warning may be issued in case of using unresolved pointers or aliases. The RiTHM user guide describes all the limitations (mostly due to LLVM capabilities) in detail.

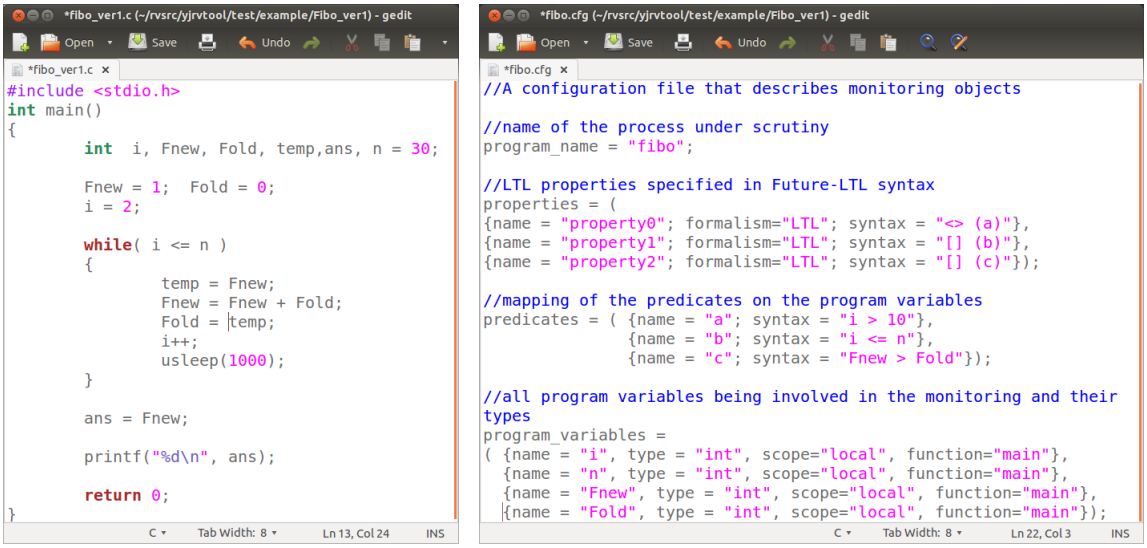
After running the tool, the instrumented program is the one shown in Figure 8.8. As can be seen, the instrumentation adds the value of the variables of interest based on the given input sampling period such that no property valuation is overlooked. It can also be seen that Globalizer has prefixed variables by the name of the function that declares the variables of interest (i.e., function main).

8.5.4 Running the Instrumented Application and Observing Online Statistics

RiTHM provides detailed report on the valuation of LTL properties and maintains a trace log that enables the user to trace back the value of variables that participate in LTL properties. This is achieved by the help of the name of the source C file and the line number of the instruction that changes the value of the variable. Figures 8.9 and 8.10 show the property and trace log of the case study. RiTHM reports that property0 has been satisfied, but property1 and property2 are violated. The trace log illustrates the reason. Property1 is violated because when the loop terminates the value of i is 31 while the value

of n is 30. Property2 is violated because the values of Fnew and Fold are zero in the initial state of the program. We note that generating the trace log is optional and it does not necessarily impose overhead.

When the dynamic sampling period is used, RiTHM provides a real-time plot of the current sampling period of the time-triggered monitor and the utilization of the available buffer as seen in Figure 8.3(b).



(a) Input C program (Fibonacci function)

(b) LTL Properties

Figure 8.6: Fibonacci function with its properties

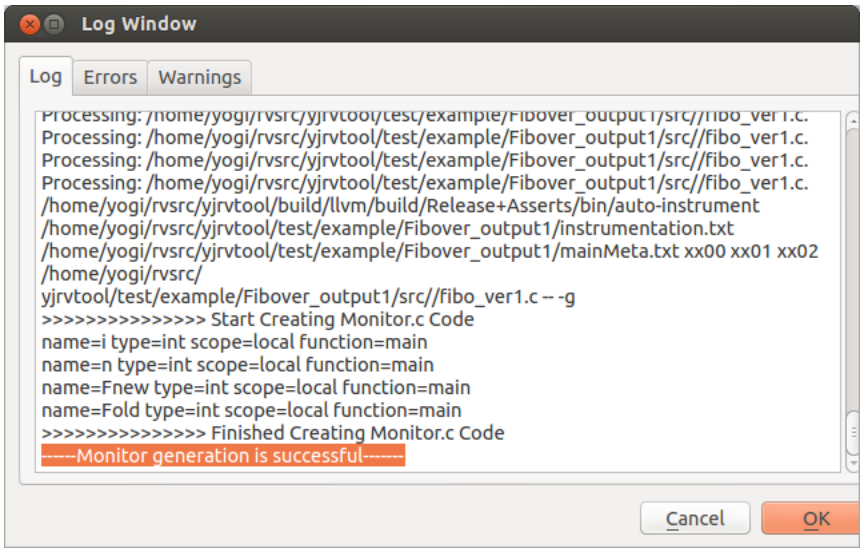


Figure 8.7: Instrumentation log

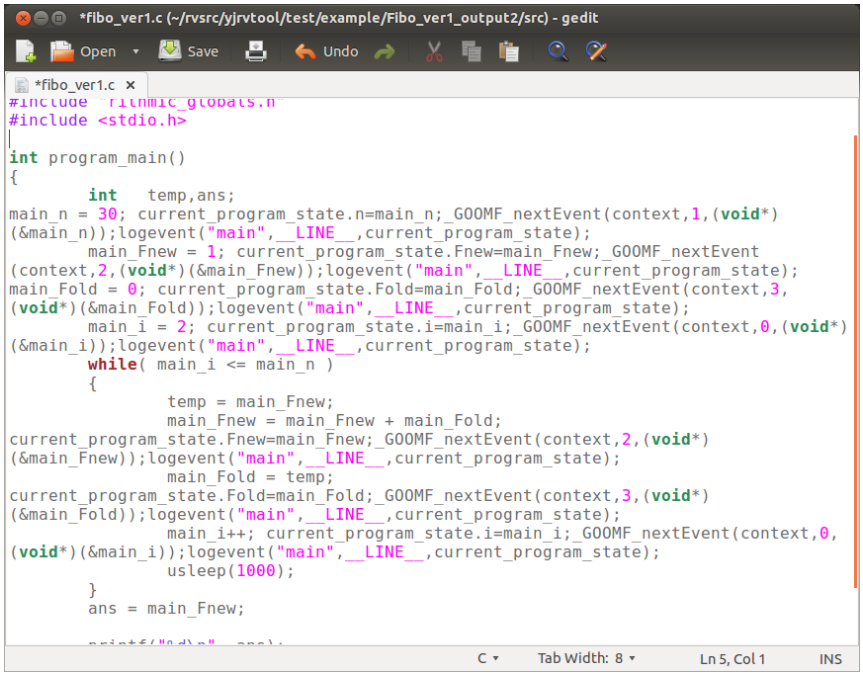


Figure 8.8: Instrumented code

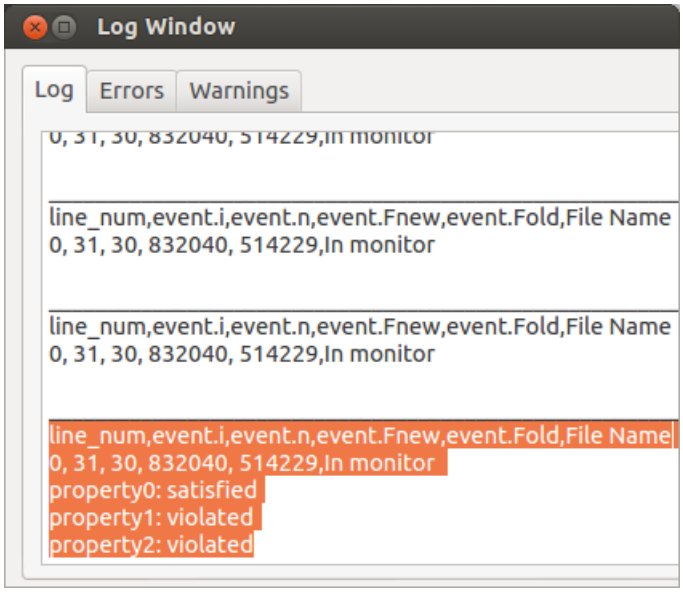


Figure 8.9: Property valuations report

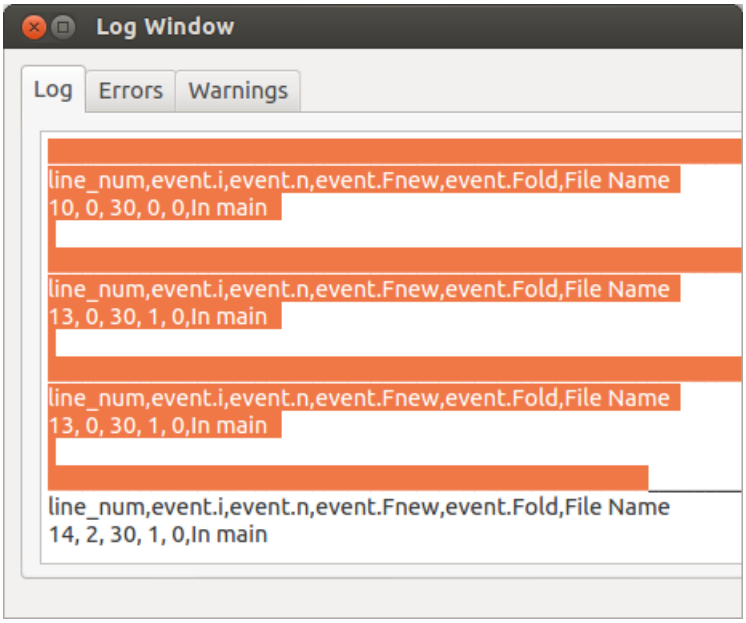


Figure 8.10: Trace log

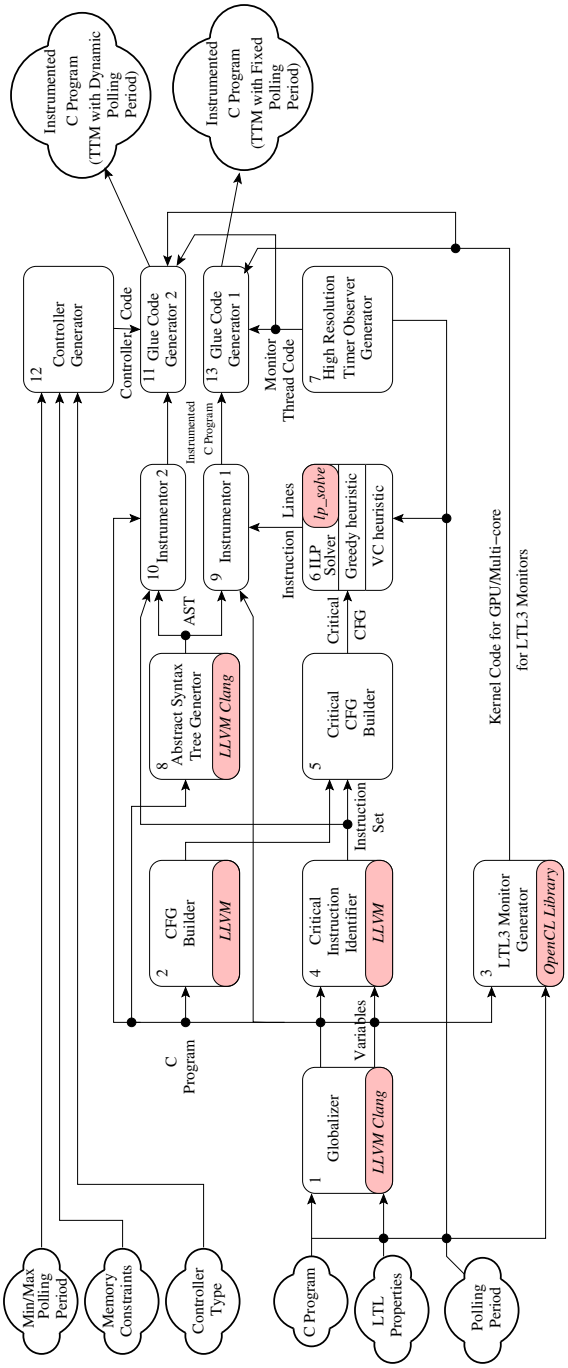


Figure 8.11: Building blocks and data flow in RiTHM.

Chapter 9

Conclusion

Ensuring the correctness of safety-critical real-time embedded systems is vital to the persistence of technology as we see it today. In the recent years, it has become evident that testing and program verification does not suffice to ensure correctness in such systems. To this end, *runtime verification* was established that verifies the behavior of the system at run time. The runtime verification framework for safety-critical real-time embedded systems must have the following characteristics to prevent transient overload situations that can result in catastrophic consequences in such systems.

1. predictable runtime monitoring,
2. bounded runtime overheard,
3. efficient runtime monitoring overhead, and
4. reduced over-provisioning.

Event-triggered monitoring has been the main monitoring technique in runtime verification. Event-triggered monitoring has jittery runtime overhead and its monitoring invocations are unevenly distributed throughout the program run. These two features cause transient overload situations at run time and impose over-provisioning of resources. To this end, event-triggered monitoring is unsuitable for safety-critical real-time embedded systems.

To best of our knowledge, a monitoring technique customized for safety-critical real-time embedded systems has not been established. To this end, in this dissertation, we

present *time-triggered* monitoring which is a novel approach suited for runtime verification of safety-critical real-time embedded systems. In time-triggered monitoring, the monitor is invoked based on a predefined time interval, where at each invocation, the monitor extracts all the program state information required to verify the runtime behavior of the program under scrutiny. With this respect, time-triggered monitoring provides: (1) predictable runtime monitoring, since the monitor invocations are evenly distributed through the program run, (2) bounded runtime overhead, since the monitor extracts the *same* set of state information for verification at each monitor invocation, and (3) because of predictable runtime monitoring and bounded runtime overhead, it reduces over-provisioning of resources.

The main issues surrounding time-triggered monitoring is:

1. Ensuring sound runtime verification, and
2. Providing efficient runtime monitoring overhead.

In this dissertation, we tackled both issues. For sound runtime verification, we established the theoretical foundation of calculating the time interval for the time-triggered monitor such that all state transitions of the program required for verifying its runtime behavior are sampled throughout the program run. We present static analysis techniques to analyze the structure and predict the runtime behavior of the program to calculate the time interval. Mathematical proof and rigorous experimental results prove the correctness and completeness of our approach. As for providing efficient runtime monitoring overhead, we present two main optimization techniques based on using auxiliary memory and predicting the execution path of the program to decrease the number of monitor invocations and hence, reduce the runtime monitoring overhead. Experimental results show that our optimization techniques successfully result in a time-triggered monitor that not only ensures sound runtime verification and efficient monitoring overhead, but also provides predictable monitoring, bounded overhead, and reduced over-provisioning.

To further advance time-triggered monitoring, we establish an optimization technique to enable efficient time-triggered monitoring of component-based embedded systems running on multi-core architectures. In addition, we gathered all our work on time-triggered monitoring into an open source stand alone tool named RiTHM that can be downloaded and used for the runtime verification of safety-critical real-time embedded systems written in C.

The aforementioned issues are general issues surrounding the runtime verification of safety-critical real-time embedded systems. This dissertation only presents a handful of solutions to overcome these issues. Hence, this dissertation can be a starting point for new

areas of research aimed at runtime verification of time-sensitive systems. This dissertation is a road map to establishing a generic time-triggered monitoring framework which can be leveraged by other runtime verification frameworks. To this end, the techniques and theoretical foundations can be further advanced in various areas and disciplines to enhance the correctness of safety-critical real-time embedded systems. Some of the future work on this subject are as follows:

1. In addition to the execution path of the program to improve the invocation time interval of the time-triggered monitor, one can incorporate the state of the property being verified to also further eliminate unnecessary monitor invocations.

2. Advance the time-triggered monitor such that it can verify the program under scrutiny with respect to metric temporal logic properties.

3. The time-triggered monitor can be power-aware. In other words, the invocation time interval of the time-triggered monitor can be adjusted at run time to fit the power resource available to the embedded system such that time-triggered monitors can be used in power sensitive systems such as mobile devices.

4. Advance the time-triggered monitor to verify properties of distributed systems. At the time being, the time-triggered monitor can only monitor programs running on a single processing unit because of the limitations regarding its current verification engine and the existence of timer skews among processing units.

5. Advance the time-triggered monitor to verify properties of hard real-time embedded systems.

Letters of Copyright Permission

Permission for use of “Sampling-Based Runtime Verification” from FM’11

SPRINGER LICENSE
TERMS AND CONDITIONS

Dec 05, 2013

This is a License Agreement between Samaneh Navabpour ("You") and Springer ("Springer") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by Springer, and the payment terms and conditions.

All payments must be made in full to CCC. For payment instructions, please see information listed at the bottom of this form.

License Number	3282551314801
License date	Dec 05, 2013
Licensed content publisher	Springer
Licensed content publication	Springer eBook
Licensed content title	Sampling-Based Runtime Verification
Licensed content author	Borzoo Bonakdarpour
Licensed content date	Jan 1, 2011
Type of Use	Thesis/Dissertation
Portion	Full text
Number of copies	1
Author of this Springer article	Yes and you are the sole author of the new work
Order reference number	
Title of your thesis / dissertation	Time-triggered Runtime Verification of Real-time Embedded Systems
Expected completion date	Jan 2014
Estimated size(pages)	170
Total	0.00 USD
Terms and Conditions	

Introduction
The publisher for this copyrighted material is Springer Science + Business Media. By clicking "accept" in connection with completing this licensing transaction, you agree that the following terms and conditions apply to this transaction (along with the Billing and Payment terms and conditions established by Copyright Clearance Center, Inc. ("CCC"), at the time that you opened your Rightslink account and that are available at any time at http://myaccount.copyright.com).

Example 1: Springer Science + Business Media makes no representations or warranties with respect to the licensed material.

Example 2: Springer Science + Business Media makes no representations or warranties with respect to the licensed material and adopts on its own behalf the limitations and disclaimers established by CCC on its behalf in its Billing and Payment terms and conditions for this licensing transaction.

Indemnity
You hereby indemnify and agree to hold harmless Springer Science + Business Media and CCC, and their respective officers, directors, employees and agents, from and against any and all claims arising out of your use of the licensed material other than as specifically authorized pursuant to this license.

No Transfer of License
This license is personal to you and may not be sublicensed, assigned, or transferred by you to any other person without Springer Science + Business Media's written permission.

No Amendment Except in Writing
This license may not be amended except in a writing signed by both parties (or, in the case of Springer Science + Business Media, by CCC on Springer Science + Business Media's behalf).

Objection to Contrary Terms
Springer Science + Business Media hereby objects to any terms contained in any purchase order, acknowledgment, check endorsement or other writing prepared by you, which terms are inconsistent with these terms and conditions or CCC's Billing and Payment terms and conditions. These terms and conditions, together with CCC's Billing and Payment terms and conditions (which are incorporated herein), comprise the entire agreement between you and Springer Science + Business Media (and CCC) concerning this licensing transaction. In the event of any conflict between your obligations established by these terms and conditions and those established by CCC's Billing and Payment terms and conditions, these terms and conditions shall control.

Jurisdiction
All disputes that may arise in connection with this present License, or the breach thereof, shall be settled exclusively by arbitration, to be held in The Netherlands, in accordance with Dutch law, and to be conducted under the Rules of the 'Netherlands Arbitrage Instituut' (Netherlands Institute of Arbitration).***OR:***

All disputes that may arise in connection with this present License, or the breach thereof, shall be settled exclusively by arbitration, to be held in the Federal Republic of Germany, in accordance with German law.

Other terms and conditions:

v1.3

If you would like to pay for this license now, please remit this license along with your

If you would like to pay for this license now, please remit this license along with your payment made payable to "COPYRIGHT CLEARANCE CENTER" otherwise you will be invoiced within 48 hours of the license date. Payment should be in the form of a check or money order referencing your account number and this invoice number RLNK501175274.
Once you receive your invoice for this order, you may pay your invoice by credit card. Please follow instructions provided at that time.

Make Payment To:
Copyright Clearance Center
Dept 001
P.O. Box 843006
Boston, MA 02284-3006

For suggestions or comments regarding this order, contact RightsLink Customer Support: customercare@copyright.com or +1-877-622-5543 (toll free in the US) or +1-978-646-2777.

Gratis licenses (referencing $0 in the Total field) are free. Please retain this printable license for your reference. No payment is required.

Permission for use of “Efficient Techniques for Near-Optimal Instrumentation in Time- Triggered Runtime Verification” from RV’11

SPRINGER LICENSE
TERMS AND CONDITIONS

Dec 05, 2013

This is a License Agreement between Samaneh Navabpour ("You") and Springer ("Springer") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by Springer, and the payment terms and conditions.

All payments must be made in full to CCC. For payment instructions, please see information listed at the bottom of this form.

License Number	3282551175175
License date	Dec 05, 2013
Licensed content publisher	Springer
Licensed content publication	Springer eBook
Licensed content title	Efficient Techniques for Near-Optimal Instrumentation in Time-Triggered Runtime Verification
Licensed content author	Samaneh Navabpour
Licensed content date	Jan 1, 2012
Type of Use	Thesis/Dissertation
Portion	Full text
Number of copies	1
Author of this Springer article	Yes and you are the sole author of the new work
Order reference number	
Title of your thesis / dissertation	Time-triggered Runtime Verification of Real-time Embedded Systems
Expected completion date	Jan 2014
Estimated size(pages)	170
Total	0.00 USD
Terms and Conditions	

Introduction
The publisher for this copyrighted material is Springer Science + Business Media. By clicking "accept" in connection with completing this licensing transaction, you agree that the following terms and conditions apply to this transaction (along with the Billing and Payment terms and conditions established by Copyright Clearance Center, Inc. ("CCC"), at the time that you opened your Rightslink account and that are available at any time at http://myaccount.copyright.com).

Limited License
With reference to your request to reprint in your thesis material on which Springer Science and Business Media control the copyright, permission is granted, free of charge, for the use indicated in your enquiry.

Licenses are for one-time use only with a maximum distribution equal to the number that you identified in the licensing process.

This License includes use in an electronic form, provided its password protected or on the university's intranet or repository, including UMI (according to the definition at the Sherpa website: http://www.sherpa.ac.uk/romeo/). For any other electronic use, please contact Springer at (permissions.dordrecht@springer.com or permissions.heidelberg@springer.com).

The material can only be used for the purpose of defending your thesis, and with a maximum of 100 extra copies in paper.

Although Springer holds copyright to the material and is entitled to negotiate on rights, this license is only valid, subject to a courtesy information to the author (address is given with the article/chapter) and provided it concerns original material which does not carry references to other sources (if material in question appears with credit to another source, authorization from that source is required as well).

Permission free of charge on this occasion does not prejudice any rights we might have to charge for reproduction of our copyrighted material in the future.

Altering/Modifying Material: Not Permitted
You may not alter or modify the material in any manner. Abbreviations, additions, deletions and/or any other alterations shall be made only with prior written authorization of the author(s) and/or Springer Science + Business Media. (Please contact Springer at (permissions.dordrecht@springer.com or permissions.heidelberg@springer.com)

Reservation of Rights
Springer Science + Business Media reserves all rights not specifically granted in the combination of (i) the license details provided by you and accepted in the course of this licensing transaction, (ii) these terms and conditions and (iii) CCC's Billing and Payment terms and conditions.

Copyright Notice:Disclaimer
You must include the following copyright and permission notice in connection with any reproduction of the licensed material: "Springer and the original publisher /journal title, volume, year of publication, page, chapter/article title, name(s) of author(s), figure number(s), original copyright notice) is given to the publication in which the material was originally published, by adding; with kind permission from Springer Science and Business Media"

Warranties: None

Example 1: Springer Science + Business Media makes no representations or warranties with respect to the licensed material.

Example 2: Springer Science + Business Media makes no representations or warranties with respect to the licensed material and adopts on its own behalf the limitations and disclaimers established by CCC on its behalf in its Billing and Payment terms and conditions for this licensing transaction.

Indemnity
You hereby indemnify and agree to hold harmless Springer Science + Business Media and CCC, and their respective officers, directors, employees and agents, from and against any and all claims arising out of your use of the licensed material other than as specifically authorized pursuant to this license.

No Transfer of License
This license is personal to you and may not be sublicensed, assigned, or transferred by you to any other person without Springer Science + Business Media's written permission.

No Amendment Except in Writing
This license may not be amended except in a writing signed by both parties (or, in the case of Springer Science + Business Media, by CCC on Springer Science + Business Media's behalf).

Objection to Contrary Terms
Springer Science + Business Media hereby objects to any terms contained in any purchase order, acknowledgment, check endorsement or other writing prepared by you, which terms are inconsistent with these terms and conditions or CCC's Billing and Payment terms and conditions. These terms and conditions, together with CCC's Billing and Payment terms and conditions (which are incorporated herein), comprise the entire agreement between you and Springer Science + Business Media (and CCC) concerning this licensing transaction. In the event of any conflict between your obligations established by these terms and conditions and those established by CCC's Billing and Payment terms and conditions, these terms and conditions shall control.

Jurisdiction
All disputes that may arise in connection with this present License, or the breach thereof, shall be settled exclusively by arbitration, to be held in The Netherlands, in accordance with Dutch law, and to be conducted under the Rules of the 'Netherlands Arbitrage Instituut' (Netherlands Institute of Arbitration).***OR:***

All disputes that may arise in connection with this present License, or the breach thereof, shall be settled exclusively by arbitration, to be held in the Federal Republic of Germany, in accordance with German law.

Other terms and conditions:

v1.3

If you would like to pay for this license now, please remit this license along with your

If you would like to pay for this license now, please remit this license along with your payment made payable to "COPYRIGHT CLEARANCE CENTER" otherwise you will be invoiced within 48 hours of the license date. Payment should be in the form of a check or money order referencing your account number and this invoice number RLNK501175268.
Once you receive your invoice for this order, you may pay your invoice by credit card. Please follow instructions provided at that time.

Make Payment To:
Copyright Clearance Center
Dept 001
P.O. Box 843006
Boston, MA 02284-3006

For suggestions or comments regarding this order, contact RightsLink Customer Support: customercare@copyright.com or +1-877-622-5543 (toll free in the US) or +1-978-646-2777.

Gratis licenses (referencing $0 in the Total field) are free. Please retain this printable license for your reference. No payment is required.

Permission for use of “Path-Aware Time-Triggered Runtime Verification” from RV’12

SPRINGER LICENSE
TERMS AND CONDITIONS

Dec 05, 2013

This is a License Agreement between Samaneh Navabpour ("You") and Springer ("Springer") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by Springer, and the payment terms and conditions.

All payments must be made in full to CCC. For payment instructions, please see information listed at the bottom of this form.

License Number	3282551032511
License date	Dec 05, 2013
Licensed content publisher	Springer
Licensed content publication	Springer eBook
Licensed content title	Path-Aware Time-Triggered Runtime Verification
Licensed content author	Samaneh Navabpour
Licensed content date	Jan 1, 2013
Type of Use	Thesis/Dissertation
Portion	Full text
Number of copies	1
Author of this Springer article	Yes and you are the sole author of the new work
Order reference number	
Title of your thesis / dissertation	Time-triggered Runtime Verification of Real-time Embedded Systems
Expected completion date	Jan 2014
Estimated size(pages)	170
Total	0.00 USD
Terms and Conditions	

Introduction
The publisher for this copyrighted material is Springer Science + Business Media. By clicking "accept" in connection with completing this licensing transaction, you agree that the following terms and conditions apply to this transaction (along with the Billing and Payment terms and conditions established by Copyright Clearance Center, Inc. ("CCC"), at the time that you opened your Rightslink account and that are available at any time at http://myaccount.copyright.com).

Limited License
With reference to your request to reprint in your thesis material on which Springer Science and Business Media control the copyright, permission is granted, free of charge, for the use indicated in your enquiry.

Licenses are for one-time use only with a maximum distribution equal to the number that you identified in the licensing process.

This License includes use in an electronic form, provided its password protected or on the university's intranet or repository, including UMI (according to the definition at the Sherpa website: http://www.sherpa.ac.uk/romeo/). For any other electronic use, please contact Springer at (permissions.dordrecht@springer.com or permissions.heidelberg@springer.com).

The material can only be used for the purpose of defending your thesis, and with a maximum of 100 extra copies in paper.

Although Springer holds copyright to the material and is entitled to negotiate on rights, this license is only valid, subject to a courtesy information to the author (address is given with the article/chapter) and provided it concerns original material which does not carry references to other sources (if material in question appears with credit to another source, authorization from that source is required as well).

Permission free of charge on this occasion does not prejudice any rights we might have to charge for reproduction of our copyrighted material in the future.

Altering/Modifying Material: Not Permitted
You may not alter or modify the material in any manner. Abbreviations, additions, deletions and/or any other alterations shall be made only with prior written authorization of the author(s) and/or Springer Science + Business Media. (Please contact Springer at (permissions.dordrecht@springer.com or permissions.heidelberg@springer.com)

Reservation of Rights
Springer Science + Business Media reserves all rights not specifically granted in the combination of (i) the license details provided by you and accepted in the course of this licensing transaction, (ii) these terms and conditions and (iii) CCC's Billing and Payment terms and conditions.

Copyright Notice:Disclaimer
You must include the following copyright and permission notice in connection with any reproduction of the licensed material: "Springer and the original publisher /journal title, volume, year of publication, page, chapter/article title, name(s) of author(s), figure number(s), original copyright notice) is given to the publication in which the material was originally published, by adding; with kind permission from Springer Science and Business Media"

Warranties: None

Example 1: Springer Science + Business Media makes no representations or warranties with respect to the licensed material.

Example 2: Springer Science + Business Media makes no representations or warranties with respect to the licensed material and adopts on its own behalf the limitations and disclaimers established by CCC on its behalf in its Billing and Payment terms and conditions for this licensing transaction.

Indemnity
You hereby indemnify and agree to hold harmless Springer Science + Business Media and CCC, and their respective officers, directors, employees and agents, from and against any and all claims arising out of your use of the licensed material other than as specifically authorized pursuant to this license.

No Transfer of License
This license is personal to you and may not be sublicensed, assigned, or transferred by you to any other person without Springer Science + Business Media's written permission.

No Amendment Except in Writing
This license may not be amended except in a writing signed by both parties (or, in the case of Springer Science + Business Media, by CCC on Springer Science + Business Media's behalf).

Objection to Contrary Terms
Springer Science + Business Media hereby objects to any terms contained in any purchase order, acknowledgment, check endorsement or other writing prepared by you, which terms are inconsistent with these terms and conditions or CCC's Billing and Payment terms and conditions. These terms and conditions, together with CCC's Billing and Payment terms and conditions (which are incorporated herein), comprise the entire agreement between you and Springer Science + Business Media (and CCC) concerning this licensing transaction. In the event of any conflict between your obligations established by these terms and conditions and those established by CCC's Billing and Payment terms and conditions, these terms and conditions shall control.

Jurisdiction
All disputes that may arise in connection with this present License, or the breach thereof, shall be settled exclusively by arbitration, to be held in The Netherlands, in accordance with Dutch law, and to be conducted under the Rules of the 'Netherlands Arbitrage Instituut' (Netherlands Institute of Arbitration).***OR:***

All disputes that may arise in connection with this present License, or the breach thereof, shall be settled exclusively by arbitration, to be held in the Federal Republic of Germany, in accordance with German law.

Other terms and conditions:

v1.3

If you would like to pay for this license now, please remit this license along with your

If you would like to pay for this license now, please remit this license along with your payment made payable to "COPYRIGHT CLEARANCE CENTER" otherwise you will be invoiced within 48 hours of the license date. Payment should be in the form of a check or money order referencing your account number and this invoice number RLNK501175264.
Once you receive your invoice for this order, you may pay your invoice by credit card. Please follow instructions provided at that time.

Make Payment To:
Copyright Clearance Center
Dept 001
P.O. Box 843006
Boston, MA 02284-3006

For suggestions or comments regarding this order, contact RightsLink Customer Support: customercare@copyright.com or +1-877-622-5543 (toll free in the US) or +1-978-646-2777.

Gratis licenses (referencing $0 in the Total field) are free. Please retain this printable license for your reference. No payment is required.

Permission for use of “RiTHM: a tool for enabling time-triggered runtime verification for C programs” from FSE’13

ASSOCIATION FOR COMPUTING MACHINERY, INC. LICENSE TERMS AND CONDITIONS

Dec 05, 2013

This is a License Agreement between Samaneh Navabpour ("You") and Association for Computing Machinery, Inc. ("Association for Computing Machinery, Inc.") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by Association for Computing Machinery, Inc., and the payment terms and conditions.

License Number	3282550465896
License date	Dec 05, 2013
Licensed content publisher	Association for Computing Machinery, Inc.
Licensed content publication	Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering
Licensed content title	RiTHM: a tool for enabling time-triggered runtime verification for C programs
Licensed content author	Samaneh Navabpour, et al
Licensed content date	Aug 18, 2013
Type of Use	Thesis/Dissertation
Requestor type	Author of this ACM article
Is reuse in the author's own new work?	No
Format	Print and electronic
Portion	Full article
Will you be translating?	No
Order reference number	
Title of your thesis/dissertation	Time-triggered Runtime Verification of Real-time Embedded Systems
Expected completion date	Jan 2014
Estimated size (pages)	170
Billing Type	Credit Card
Credit card info	Visa ending in 4076
Credit card expiration	05/2016
Total	8.00 USD
Terms and Conditions	

Rightslink Terms and Conditions for ACM Material

1. The publisher of this copyrighted material is Association for Computing Machinery, Inc. (ACM). By clicking "accept" in connection with completing this licensing transaction, you agree that the following terms and conditions apply to this transaction (along with the Billing and Payment terms and conditions established by Copyright Clearance Center, Inc. ("CCC"), at the time that you opened your Rightslink account and that are available at any time at).

2. ACM reserves all rights not specifically granted in the combination of (i) the license details provided by you and accepted in the course of this licensing transaction, (ii) these terms and conditions and (iii) CCC's Billing and Payment terms and conditions.

3. ACM hereby grants to licensee a non-exclusive license to use or republish this ACM-copyrighted material* in secondary works (especially for commercial distribution) with the stipulation that consent of the lead author has been obtained independently. Unless otherwise stipulated in a license, grants are for one-time use in a single edition of the work, only with a maximum distribution equal to the number that you identified in the licensing process. Any additional form of republication must be specified according to the terms included at the time of licensing.

*Please note that ACM cannot grant republication or distribution licenses for embedded third-party material. You must confirm the ownership of figures, drawings and artwork prior to use.

4. Any form of republication or redistribution must be used within 180 days from the date stated on the license and any electronic posting is limited to a period of six months unless an extended term is selected during the licensing process. Separate subsidiary and subsequent republication licenses must be purchased to redistribute copyrighted material on an extranet. These licenses may be exercised anywhere in the world.

5. Licensee may not alter or modify the material in any manner (except that you may use, within the scope of the license granted, one or more excerpts from the copyrighted material, provided that the process of excerpting does not alter the meaning of the material or in any way reflect negatively on the publisher or any writer of the material).

6. Licensee must include the following copyright and permission notice in connection with any reproduction of the licensed material: "[Citation] © YEAR Association for Computing Machinery, Inc. Reprinted by permission." Include the article DOI as a link to the definitive version in the ACM Digital Library. Example: Charles, L. "How to Improve Digital Rights Management," Communications of the ACM, Vol. 51:12, © 2008 ACM, Inc. http://doi.acm.org/10.1145/nnnnnn.nnnnnn (where nnnnnn.nnnnnn is replaced by the actual number).

7. Translation of the material in any language requires an explicit license identified during the licensing process. Due to the error-prone nature of language translations, Licensee must include the following copyright and permission notice and disclaimer in connection with any reproduction of the licensed material in translation: "This translation is a derivative of ACM-copyrighted material. ACM did not prepare this translation and does not guarantee that it is

an accurate copy of the originally published work. The original intellectual property contained in this work remains the property of ACM."

8. You may exercise the rights licensed immediately upon issuance of the license at the end of the licensing transaction, provided that you have disclosed complete and accurate details of your proposed use. No license is finally effective unless and until full payment is received from you (either by CCC or ACM) as provided in CCC's Billing and Payment terms and conditions.

9. If full payment is not received within 90 days from the grant of license transaction, then any license preliminarily granted shall be deemed automatically revoked and shall be void as if never granted. Further, in the event that you breach any of these terms and conditions or any of CCC's Billing and Payment terms and conditions, the license is automatically revoked and shall be void as if never granted.

10. Use of materials as described in a revoked license, as well as any use of the materials beyond the scope of an unrevoked license, may constitute copyright infringement and publisher reserves the right to take any and all action to protect its copyright in the materials.

11. ACM makes no representations or warranties with respect to the licensed material and adopts on its own behalf the limitations and disclaimers established by CCC on its behalf in its Billing and Payment terms and conditions for this licensing transaction.

12. You hereby indemnify and agree to hold harmless ACM and CCC, and their respective officers, directors, employees and agents, from and against any and all claims arising out of your use of the licensed material other than as specifically authorized pursuant to this license.

13. This license is personal to the requestor and may not be sublicensed, assigned, or transferred by you to any other person without publisher's written permission.

14. This license may not be amended except in a writing signed by both parties (or, in the case of ACM, by CCC on its behalf).

15. ACM hereby objects to any terms contained in any purchase order, acknowledgment, check endorsement or other writing prepared by you, which terms are inconsistent with these terms and conditions or CCC's Billing and Payment terms and conditions. These terms and conditions, together with CCC's Billing and Payment terms and conditions (which are incorporated herein), comprise the entire agreement between you and ACM (and CCC) concerning this licensing transaction. In the event of any conflict between your obligations established by these terms and conditions and those established by CCC's Billing and Payment terms and conditions, these terms and conditions shall control.

16. This license transaction shall be governed by and construed in accordance with the laws of New York State. You hereby agree to submit to the jurisdiction of the federal and state courts located in New York for purposes of resolving any disputes that may arise in connection with this licensing transaction.

17. There are additional terms and conditions, established by Copyright Clearance Center, Inc. ("CCC") as the administrator of this licensing service that relate to billing and payment for licenses provided through this service. Those terms and conditions apply to each transaction as if they were restated here. As a user of this service, you agreed to those terms and conditions at the time that you established your account, and you may see them again at any time at http://myaccount.copyright.com

18. Thesis/Dissertation: This type of use requires only the minimum administrative fee. It is not a fee for permission. Further reuse of ACM content, by ProQuest/UMI or other document delivery providers, or in republication requires a separate permission license and fee. Commercial resellers of your dissertation containing this article must acquire a separate license.

Special Terms:

If you would like to pay for this license now, please remit this license along with your payment made payable to "COPYRIGHT CLEARANCE CENTER" otherwise you will be invoiced within 48 hours of the license date. Payment should be in the form of a check or money order referencing your account number and this invoice number RLNK501175251.
Once you receive your invoice for this order, you may pay your invoice by credit card. Please follow instructions provided at that time.

Make Payment To:
Copyright Clearance Center
Dept 001
P.O. Box 843006
Boston, MA 02284-3006

For suggestions or comments regarding this order, contact RightsLink Customer Support: customercare@copyright.com or +1-877-622-5543 (toll free in the US) or +1-978-646-2777.

Gratis licenses (referencing $0 in the Total field) are free. Please retain this printable license for your reference. No payment is required.

Permission for use of “Time-triggered Runtime Verification” from FMSD’13

SPRINGER LICENSE
TERMS AND CONDITIONS

Dec 05, 2013

This is a License Agreement between Samaneh Navabpour ("You") and Springer ("Springer") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by Springer, and the payment terms and conditions.

All payments must be made in full to CCC. For payment instructions, please see information listed at the bottom of this form.

License Number	3282550891516
License date	Dec 05, 2013
Licensed content publisher	Springer
Licensed content publication	Formal Methods in System Design
Licensed content title	Time-triggered runtime verification
Licensed content author	Borzoo Bonakdarpour
Licensed content date	Jan 1, 2013
Volume number	43
Issue number	1
Type of Use	Thesis/Dissertation
Portion	Full text
Number of copies	1
Author of this Springer article	Yes and you are the sole author of the new work
Order reference number	
Title of your thesis / dissertation	Time-triggered Runtime Verification of Real-time Embedded Systems
Expected completion date	Jan 2014
Estimated size(pages)	170
Total	0.00 USD
Terms and Conditions	

Introduction
The publisher for this copyrighted material is Springer Science + Business Media. By clicking "accept" in connection with completing this licensing transaction, you agree that the following terms and conditions apply to this transaction (along with the Billing and Payment terms and conditions established by Copyright Clearance Center, Inc. ("CCC"), at the time

that you opened your Rightslink account and that are available at any time at http://myaccount.copyright.com).

Limited License
With reference to your request to reprint in your thesis material on which Springer Science and Business Media control the copyright, permission is granted, free of charge, for the use indicated in your enquiry.

Licenses are for one-time use only with a maximum distribution equal to the number that you identified in the licensing process.

This License includes use in an electronic form, provided its password protected or on the university's intranet or repository, including UMI (according to the definition at the Sherpa website: http://www.sherpa.ac.uk/romeo/). For any other electronic use, please contact Springer at (permissions.dordrecht@springer.com or permissions.heidelberg@springer.com).

The material can only be used for the purpose of defending your thesis, and with a maximum of 100 extra copies in paper.

Although Springer holds copyright to the material and is entitled to negotiate on rights, this license is only valid, subject to a courtesy information to the author (address is given with the article/chapter) and provided it concerns original material which does not carry references to other sources (if material in question appears with credit to another source, authorization from that source is required as well).

Permission free of charge on this occasion does not prejudice any rights we might have to charge for reproduction of our copyrighted material in the future.

Altering/Modifying Material: Not Permitted
You may not alter or modify the material in any manner. Abbreviations, additions, deletions and/or any other alterations shall be made only with prior written authorization of the author(s) and/or Springer Science + Business Media. (Please contact Springer at (permissions.dordrecht@springer.com or permissions.heidelberg@springer.com)

Reservation of Rights
Springer Science + Business Media reserves all rights not specifically granted in the combination of (i) the license details provided by you and accepted in the course of this licensing transaction, (ii) these terms and conditions and (iii) CCC's Billing and Payment terms and conditions.

Copyright Notice:Disclaimer
You must include the following copyright and permission notice in connection with any reproduction of the licensed material: "Springer and the original publisher /journal title, volume, year of publication, page, chapter/article title, name(s) of author(s), figure number(s), original copyright notice) is given to the publication in which the material was originally published, by adding; with kind permission from Springer Science and Business Media"

Warranties: None

Example 1: Springer Science + Business Media makes no representations or warranties with respect to the licensed material.

Example 2: Springer Science + Business Media makes no representations or warranties with respect to the licensed material and adopts on its own behalf the limitations and disclaimers established by CCC on its behalf in its Billing and Payment terms and conditions for this licensing transaction.

Indemnity
You hereby indemnify and agree to hold harmless Springer Science + Business Media and CCC, and their respective officers, directors, employees and agents, from and against any and all claims arising out of your use of the licensed material other than as specifically authorized pursuant to this license.

No Transfer of License
This license is personal to you and may not be sublicensed, assigned, or transferred by you to any other person without Springer Science + Business Media's written permission.

No Amendment Except in Writing
This license may not be amended except in a writing signed by both parties (or, in the case of Springer Science + Business Media, by CCC on Springer Science + Business Media's behalf).

Objection to Contrary Terms
Springer Science + Business Media hereby objects to any terms contained in any purchase order, acknowledgment, check endorsement or other writing prepared by you, which terms are inconsistent with these terms and conditions or CCC's Billing and Payment terms and conditions. These terms and conditions, together with CCC's Billing and Payment terms and conditions (which are incorporated herein), comprise the entire agreement between you and Springer Science + Business Media (and CCC) concerning this licensing transaction. In the event of any conflict between your obligations established by these terms and conditions and those established by CCC's Billing and Payment terms and conditions, these terms and conditions shall control.

Jurisdiction
All disputes that may arise in connection with this present License, or the breach thereof, shall be settled exclusively by arbitration, to be held in The Netherlands, in accordance with Dutch law, and to be conducted under the Rules of the 'Netherlands Arbitrage Instituut' (Netherlands Institute of Arbitration).***OR:***

All disputes that may arise in connection with this present License, or the breach thereof, shall be settled exclusively by arbitration, to be held in the Federal Republic of Germany, in accordance with German law.

Other terms and conditions:

v1.3

If you would like to pay for this license now, please remit this license along with your payment made payable to "COPYRIGHT CLEARANCE CENTER" otherwise you will be invoiced within 48 hours of the license date. Payment should be in the form of a check or money order referencing your account number and this invoice number RLNK501175262.
Once you receive your invoice for this order, you may pay your invoice by credit card. Please follow instructions provided at that time.

Make Payment To:
Copyright Clearance Center
Dept 001
P.O. Box 843006
Boston, MA 02284-3006

For suggestions or comments regarding this order, contact RightsLink Customer Support: customercare@copyright.com or +1-877-622-5543 (toll free in the US) or +1-978-646-2777.

Gratis licenses (referencing $0 in the Total field) are free. Please retain this printable license for your reference. No payment is required.

APPENDICES

Appendix A

Statement of Contributions

A.1 Publications

The following is the list of publication which I have co-authored and have made use of in this desertion. For each publication, I have presented the list of my contributions.

- Software Debugging and Testing using the Abstract Diagnosis Theory, S. Navabpour and B. Bonakdarpour and S. Fischmeister, *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, Chicago, USA, pp. 111-120, April, 2011 [81].

  Percentage of contribution: 60%

  1. Defined the technique on how to calculate observability and controllability.
  2. Took part in defining the optimization problem of minimizing instrumentation to achieve observability and controllability.
  3. Defined the Integer Linear Programming (ILP) solution to the optimization problem.
  4. Implemented the toolchain.
  5. Carried out the experiments.
  6. Analyzed the experimental results.
  7. Took part in writing the paper.

- Optimal Instrumentation of Data-flow in Concurrent Data Structures, S. Navabpour and B. Bonakdarpour and S. Fischmeister, *Proceedings of the 15th International Conference On Principles Of Distributed Systems (OPODIS)*, Toulouse, France, pp. 497-512, December, 2011 [80].

  Percentage of contribution: 60%

  1. Formally defined the concept of observability for concurrent programs.
  2. Defined the technique on how to calculate observability in concurrent programs.
  3. Defined the SAT solution to the optimization problem.
  4. Implemented the toolchain.
  5. Carried out the experiments.
  6. Analyzed the experimental results.
  7. Took part in writing the paper.

- Sampling-based Runtime Verification, B. Bonakdarpour, S. Navabpour, and S. Fischmeister, *Proceedings of the 17th International Conference on Formal Methods (FM)*, Limerick, Ireland, pp. 88-102, June, 2011 [21].

  Percentage of contribution: 50%

  1. Took part in defining the longest sampling period.
  2. Took part in coming up with the technique in practically calculating the longest sampling period.
  3. Took part in defining the history approach.
  4. Defined the Integer Linear Programming (ILP) solution to the optimization problem of the history approach.
  5. Implemented the toolchain.
  6. Carried out the experiments.
  7. Analyzed the experimental results.
  8. Took part in writing the paper.

- Efficient Techniques for Near-optimal Instrumentation in Time-triggered Runtime Verification, S. Navabpour, C. Wah Wallac Wu, B. Bonakdarpour, and S. Fischmeister, *Proceedings of the 2nd International Conference on Runtime Verification (RV)*, San Francisco, USA, pp. 208-222, September, 2011 [85].

  Percentage of contribution: 30%

1. Defined the Genetic Algorithm approach solution to the optimization problem of the history approach.
2. Took part in the implemented the toolchain (Specifically the Genetic Algorithm approach).
3. Took part in carrying out the experiments.
4. Took part in analyzing the experimental results.
5. Took part in writing the paper.

- Path-aware Time-triggered Runtime Verification, S. Navabpour, B. Bonakdarpour, and S. Fischmeister, *Proceedings of the Third International Conference on Runtime Verification (RV)*,Istanbul, Turkey, pp. 199-213, September, 2012 citenbf12.

  Percentage of contribution: 80%

  1. Came up with the idea of using path-aware approach using symbolic execution.
  2. Defined the path-aware and adaptive path-aware solutions to decreasing redundant samples.
  3. Formalized the path-aware and adaptive path-aware solutions.
  4. Implemented the toolchain.
  5. Carried out the experiments.
  6. Analyzed the experimental results.
  7. Took part in writing the paper.

- RiTHM: A Tool for Enabling Time-triggered Runtime Verification for C Programs, S. Navabpour, Y. Joshi, C. Wah Wallace, S. Berkovich, R. Medhat, B. Bonakdarpour, and S. Fischmeister, *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE)*, St. Petersburg, Russia, pp. 603-606, August, 2013 [84].

  Percentage of contribution: 30%

  1. Defined the static longest sampling period technique.
  2. Implemented the technique to calculate the static longest period technique in Rithm.
  3. Carried out the experiments.
  4. Analyzed the experimental results.
  5. Took part in the testing and debugging of Rithm.

6. Took part in writing the paper.

- Time-triggered Runtime Verification of Component-Based Multi-core Systems, S. Navabpour, B. Bonakdarpour, and S. Fischmeister, Technical Report, University of Waterloo, 2013 [83].

  Percentage of contribution: 70%

  1. Defined the optimization problem for minimizing runtime overhead in component-based multi-core systems.
  2. Defined the technique on how to calculate the runtime overhead of a time-triggered monitor in component-based multi-core systems.
  3. Defined the Integer Linear Programming (ILP) solution to the optimization problem.
  4. Implemented the toolchain.
  5. Carried out the experiments.
  6. Analyzed the experimental results.
  7. Took part in writing the paper.

- Time-triggered Runtime Verification, B. Bonakdarpour, S. Navabpour, and S. Fischmeister, *Formal Methods in System Design (FMSD)*, vol. 43, issue 1, pp. 29–60, 2013 [19].

  Percentage of contribution: 50%

  1. Took part in defining the longest sampling period.
  2. Took part in coming up with the technique in practically calculating the longest sampling period.
  3. Took part in defining the history approach.
  4. Defined the Integer Linear Programming (ILP) solution to the optimization problem of the history approach.
  5. Implemented the toolchain.
  6. Carried out the experiments.
  7. Analyzed the experimental results.
  8. Took part in writing the paper.

A.2 Approvals

The use of the published material mentioned in Section A.1 which have been presented in this dissertation, has been approved by all the co-authors.

Sebastian Fischmeister: Approved on November 17 2013 [Signature sealed]

Borzoo Bonakdarpour: Approved on December 09 2013 [Signature sealed]

Wallace Wu: See attached email.

Shay Berkovich: See attached email.

Ramy Medhat: Approved on November 18 2013 [Signature sealed]

Yogi Joshi: Approved on November 18 2013 [Signature sealed]

From: Wallace Wu
Date: November 17, 2013 4:17 PM EST
To: Sam Navabpour
Subject: **Approval**

Hi Samaneh,

I approve the use of content from our paper, “RiTHM: a tool for enabling time-triggered runtime verification for C programs” and “Efficient Techniques in Near-optimal Instrumentation in Time-triggered Runtime Verification” in your thesis.

-Wallace Wu

From: Shay Berkovich
Date: November 18, 2013 2:10:25 PM EST
To: Sam Navabpour
Subject: **Approval**

Dear Samaneh,

As one of the authors of the following paper: "RiTHM: a tool for enabling time-triggered runtime verification for C programs" - I approve the use of its contents in your Ph.D thesys.

Best,
Shay Berkovich

References

[1] Mi Bench: A free, commercially representative embedded benchmark suite. http://www.eecs.umich.edu/mibench/.

[2] SNU Real-Time Benchmarks. http://www.cprover.org/goto-cc/examples/snu.html.

[3] STP Ccontraint Solver. https://sites.google.com/site/stpfastprover/.

[4] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[5] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming Language Design and Implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. ACM.

[6] Cyrille Artho, Doron Drusinksy, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Grigore Rosu, and Willem Visser. Experiments with test case generation and runtime analysis. In *Proceedings of the 10th international conference on Advances in Theory and Practice of Abstract State Machines*, ASM'03, pages 87–108, Berlin, Heidelberg, 2003. Springer-Verlag.

[7] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *5th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'04, pages 44–57. Springer, 2004.

[8] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 4337(6):260–272, 2006.

[9] A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14, 2011.

[10] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Proceedings of the 7th international conference on Runtime Verification*, RV'07, pages 126–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] S. Berkovich. Parallel runtime verification. Masters thesis, University of Waterloo, 2013.

[12] S. Berkovich, B. Bonakdarpour, and S. Fischmeister. Gpu-based runtime verification. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1025–1036, 2013.

[13] S. Berkovich, B. Bonakdarpour, and S. Fischmeister. GPU-based runtime verification. In *IEEE International Parallel and Distributed Processing Symposium*, IPDPS'13, pages 1025–1036, 2013.

[14] Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A. Naeem. Collaborative runtime verification with tracematches. In *Proceedings of the 7th international conference on Runtime Verification*, RV'07, pages 22–37. Springer-Verlag, 2007.

[15] Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *Proceedings of the 21st European conference on Object-Oriented Programming*, ECOOP'07, pages 525–549. Springer, 2007.

[16] Eric Bodden, Patrick Lam, and Laurie Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT international symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 36–47. ACM, 2008.

[17] Eric Bodden, Patrick Lam, and Laurie Hendren. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In *Proceedings First international conference on Runtime Verification*, RV'10, pages 183–197. Springer-Verlag, 2010.

[18] B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Sampling-based runtime verification. In *Formal Methods (FM)*, pages 88–102, 2011.

[19] B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Time-triggered runtime verification. *Formal Methods in System Design (FMSD)*, 43(1):29–60, 2013.

[20] B. Bonakdarpour, J. Thomas, and S. Fischmeister. Time-triggered self-monitoring programs. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 260–269, 2012.

[21] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. Sampling-based runtime verification. In *Formal Methods*, FM'11, pages 88–102, 2011.

[22] Mark Brorkens and Michael Mller. Dynamic event generation for runtime checking using the jdi. *Electr. Notes Theor. Comput. Sci.*, 70:21–35, 2002.

[23] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Comput. Appl.*, 14:317–329, November 2000.

[24] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, 2008.

[25] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[26] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[27] Feng Chen and Grigore Roşu. MOP: an efficient and generic runtime verification framework. *SIGPLAN Not.*, 42:569–588, 2007.

[28] Feng Chen and Grigore Rosu. Java-MOP: A monitoring oriented programming environment for java. In *Proceedings of the 11th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 546–550. Springer, 2005.

[29] Kalyan-Ram Chilakamarri and Sebastian Elbaum. Reducing coverage collection overhead with disposable instrumentation. In *Proceedings of the 15th international symposium on Software Reliability Engineering*, pages 233–244, Washington, DC, USA, 2004. IEEE Computer Society.

[30] Thomas H Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[31] P. David Coward. Symbolic execution systems - a review. *Journal of Software Engineering*, 3:229–239, November 1988.

[32] Nelly Delgado, Ann Quiroz Gates, and Steve Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transaction on Software Engineering*, 30:859–872, December 2004.

[33] Matthew B.. Dwyer, Madeline Diep, and Sebastian Elbaum. Reducing the cost of path property monitoring through sampling. In *Proceedings of the 2008 23rd IEEE/ACM international conference on Automated Software Engineering*, ASE '08, pages 228–237, Washington, DC, USA, 2008. IEEE Computer Society.

[34] Matthew B. Dwyer, Alex Kinneer, and Sebastian Elbaum. Adaptive online program analysis. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 220–229, Washington, DC, USA, 2007. IEEE Computer Society.

[35] Matthew B. Dwyer and Rahul Purandare. Residual dynamic typestate analysis exploiting static analysis: results to reformulate and reduce the cost of dynamic analysis. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 124–133, New York, NY, USA, 2007. ACM.

[36] A. Easwaran, I. Shin, O. Sokolsky, and I. Lee. Schedulability analysis of hierarchical real-time systems. Technical report, University of Pennsylvania, 2007.

[37] EEMBC. http://www.eembc.org/.

[38] Sebastian Elbaum and Madeline Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Transactions on Software Engineering*, 31(4):312–327, 2005.

[39] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[40] Long Fei and Samuel P. Midkiff. Artemis: practical runtime monitoring of applications for execution anomalies. *SIGPLAN Not.*, 41:84–95, June 2006.

[41] Long Fei and Samuel P. Midkiff. Artemis: Practical runtime monitoring of applications for execution anomalies. *SIGPLAN Not.*, 41(6):84–95, 2006.

[42] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[43] GNU debugger. http://www.gnu.org/software/gdb/.

[44] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17:120–126, June 1982.

[45] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *IEEE international Workshop on Workload Characterization (WWC)*, pages 3–14, 2001.

[46] Reiner Hähnle, Marcus Baum, Richard Bubel, and Marcel Rothe. A visual interactive debugger based on symbolic execution. In *Proceedings of the IEEE/ACM international conference on Automated Software Engineering*, ASE '10, pages 143–146, New York, NY, USA, 2010. ACM.

[47] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, New York, NY, USA, 2002. ACM.

[48] Klaus Havelund. Runtime verification of C programs. In *Proceedings of the 20th IFIP TC 6/WG 6.1 international conference on Testing of Software and Communicating Systems: 8th International Workshop*, TestCom '08 / FATES '08, pages 7–22. Springer-Verlag, 2008.

[49] Klaus Havelund and Allen Goldberg. Verified software: theories, tools, experiments. chapter Verify Your Runs, pages 374–383. Springer-Verlag, Berlin, Heidelberg, 2008.

[50] Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technologies Transfer*, 6:158–173, 2004.

[51] Klaus Havelund and Grigore Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods System Design*, 24:189–215, 2004.

[52] Jochen Hipp, Ulrich Güntzer, and Gholamreza Nakhaeizadeh. Algorithms for association rule mining – a general survey and comparison. *SIGKDD Explor. Newsl.*, 2:58–64, June 2000.

[53] X. Huang. *Compiler-assisted software model checking and monitoring*. Phd thesis, Stony Brook University, 2012.

[54] Xiaowan Huang, Justin Seyster, Sean Callanan, Ketan Dixit, Radu Grosu, Scott A. Smolka, Scott D. Stoller, and Erez Zadok. Software monitoring with controllable overhead. *International Journal on Software Tools for Technology Transfer*, 2011.

[55] Thomas Jaksch, Ronald Ortner, and Peter Auer. Near-optimal regret bounds for reinforcement learning. *Journal of Machine Learning Res.*, 99:1563–1600, August 2010.

[56] R. M. Karp. Reducibility Among Combinatorial Problems. In *Symposium on Complexity of Computer Computations*, pages 85–103, 1972.

[57] Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. Reducing configurations to monitor in a software product line. In *Proceedings of the First international conference on Runtime Verification*, RV'10, pages 285–299. Springer-Verlag, 2010.

[58] Moonjoo Kim, Insup Lee, Usa Sammapun, Jangwoo Shin, and Oleg Sokolsky. Monitoring, checking, and steering of real-time systems. *Electr. Notes Theor. Comput. Sci.*, 70(4), 2002.

[59] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A run-time assurance approach for java programs. *Formal Methods on System Design*, 24:129–155, 2004.

[60] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[61] James C. King. Symbolic execution and program testing. *Communications of ACM*, 19:385–394, July 1976.

[62] Alex Kinneer, M Dwyer, and Gregg Rothermel. Sofya : A flexible framework for development of dynamic program analyses for java software. *University of NebraskaLincoln Tech Rep TRUNLCSE20060006*, pages 1–22, 2006.

[63] Alex Kinneer, Matthew B. Dwyer, and Gregg Rothermel. Sofya : A flexible framework for development of dynamic program analyses for java software. *University of Nebraska Lincoln Technical Report TRUNLCSE20060006*, pages 1–22, April 2006.

[64] C Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization: Feedback Directed and Runtime Optimization*, page 75, 2004.

[65] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code Generation and Optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[66] I. Lee, J. Y-T. Leung, and S. H. Son. *Handbook of Real-Time and Embedded Systems*. Chapman & Hall/CRC Computer & Information Science Series, 2007.

[67] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, PLDI '03, pages 141–154, New York, NY, USA, 2003. ACM.

[68] Hui Liu, Limei Yang, Zhendong Niu, Zhyi Ma, and Weizhong Shao. Facilitating software refactoring with appropriate resolution order of bad smells. In *Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, ESEC/FSE '09, pages 265–268, New York, NY, USA, 2009. ACM.

[69] F. Long. *Practical runtime identification of program errors*. Phd thesis, Purdue University, 2006.

[70] ILP solver lp_solve. http://lpsolve.sourceforge.net/5.5/.

[71] Jonas Maebe, Michiel Ronsse, and Koen De Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Proceedings of the 4th Workshop on Binary Translation*, WBT02, 2002.

[72] Matrix TCL Library. http://www.techsoftpl.com/matrix/.

[73] R. Medhat, D. Kumar, B. Bonakdarpour, and S. Fischmeister. Runtime verification with controllable time predictability and memory utilization. Technical Report CS-2013-02, University of Waterloo, 2013.

[74] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Software Engineering*, 30:126–139, February 2004.

[75] Patrick Meredith and Grigore Roşu. Runtime verification with the RV system. In *Proceedings of the 1st international conference on Runtime Verification*, RV'10, pages 136–152, Berlin, Heidelberg, 2010. Springer-Verlag.

[76] Jonathan Misurda, James A. Clause, Juliya L. Reed, Bruce R. Childers, and Mary Lou Soffa. Demand-driven structural testing with dynamic instrumentation. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 156–165, New York, NY, USA, 2005. ACM.

[77] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press., Cambridge, MA, USA, 1998.

[78] Maja Machine Learning Framework. http://mmlf.sourceforge.net/.

[79] Emerson Murphy-Hill and Andrew P. Black. Refactoring tools: Fitness for purpose. *IEEE Transaction on Software Engineering*, 25:38–44, September 2008.

[80] S. Navabpour, B. Bonakdarpour, and S. Fischmeister. Optimal instrumentation of data-flow in concurrent data structures. In *Proceedings of the 15th International Conference On Principles Of Distributed Systems (OPODIS)*, pages 497–512, 2011.

[81] S. Navabpour, B. Bonakdarpour, and S. Fischmeister. Software debugging and testing using the abstract diagnosis theory. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, pages 111–120, 2011.

[82] S. Navabpour, B. Bonakdarpour, and S. Fischmeister. Path-aware time-triggered runtime verification. In *Runtime Verification (RV)*, pages 199–213, 2012.

[83] S. Navabpour, B. Bonakdarpour, and S. Fischmeister. Time-triggered runtime verification of component-based multi-core systems. Technical report, University of Waterloo, 2013.

[84] S. Navabpour, Y. Joshi, C. W. W. Wu, S. Berkovich, R. Medhat, B. Bonakdarpour, and S. Fischmeister. Rithm: a tool for enabling time-triggered runtime verification for c programs. In *Proceedings of the 9'th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, FSE'13, pages 603–606, 2013.

[85] Samaneh Navabpour, Wallace Wu, Borzoo Bonakdarpour, and Sebastian Fischmeister. Efficient techniques for near-optimal instrumentation in time-triggered runtime verification. In *Proc. of the 2nd international conference on Runtime Verification*, RV'11, San Francisco, USA, September 2011.

[86] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th international conference on Compiler Construction*, CC '02, pages 213–228, London, UK, 2002. Springer-Verlag.

[87] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42:89–100, June 2007.

[88] NIST. Software errors cost U.S. economy $59.5 billion annually. In *NIST News Release*, 2002.

[89] Athanassios Papagelis and Dimitrios Kalles. Breeding decision trees using evolutionary techniques. In *Proceedings of the 18th International Conference on Machine Learning*, ICML '01, pages 393–400, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[90] A. Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.

[91] Corina S. Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

[92] R. A. Purandare. *Exploiting Program and Property Structure for Efficient Runtime Monitoring*. Phd thesis, University of Nebraska - Lincoln, 2011.

[93] Lior Rokach and Oded Maimon. Top-down induction of decision trees classifiers - a survey. *IEEE Transactions on Systems, Man, and Cybernetics*, 35:476–487, November 2005.

[94] Usa Sammapun, Insup Lee, and Oleg Sokolsky. RT-MaC: Runtime monitoring and checking of quantitative and probabilistic properties. In *Proceedings of the 11th IEEE international conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '05, pages 147–153, Washington, DC, USA, 2005. IEEE Computer Society.

[95] Tom Schaul, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke, Thomas Rückstieß, and Jürgen Schmidhuber. PyBrain. *Journal of Machine Learning Research*, 11:743–746, 2010.

[96] Justin Seyster, Ketan Dixit, Xiaowan Huang, Radu Grosu, Klaus Havelund, Scott A. Smolka, Scott D. Stoller, and Erez Zadok. Aspect-oriented instrumentation with GCC. In *Proceedings of the 1st international conference on Runtime Verification*, RV'10, pages 405–420, Berlin, Heidelberg, 2010. Springer-Verlag.

[97] SHARCNET. https://www.sharcnet.ca/my/front/.

[98] SPEC 2000. http://www.spec.org/.

[99] Scott D. Stoller, Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, Scott A. Smolka, and Erez Zadok. Runtime verification with state estimation. In *Proceedings of 2nd international conference on Runtime Verification*, RV'11, pages 193–207. Springer-Verlag, September 2012.

[100] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press., Cambridge, MA, USA, 1998.

[101] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[102] TeachingBox. http://servicerobotik.hs-weingarten.de/en/teachingbox.php.

[103] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 86–96, New York, NY, USA, 2002. ACM.

[104] C. W. W. Wu. Methods for reducing monitoring overhead in runtime verification. Masters thesis, University of Waterloo, 2013.